The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report Deduplication

Joshua Charles Campbell
Department of Computing Science
University of Alberta
Edmonton, Canada
joshua2@ualberta.ca

Eddie Antonio Santos
Department of Computing Science
University of Alberta
Edmonton, Canada
eas Santos@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
hindle1@ualberta.ca

ABSTRACT
Organizations like Mozilla, Microsoft, and Apple are flooded with thousands of automated crash reports per day. Although crash reports contain valuable information for debugging, there are often too many for developers to examine individually. Therefore, in industry, crash reports are often automatically grouped together in buckets. Ubuntu’s repository contains crashes from hundreds of software systems available with Ubuntu. A variety of crash report bucketing methods are evaluated using data collected by Ubuntu’s Apport automated crash reporting system. The trade-off between precision and recall of numerous scalable crash deduplication techniques is explored. A set of criteria that a crash deduplication method must meet is presented and several methods that meet these criteria are evaluated on a new dataset. The evaluations presented in this paper show that using off-the-shelf information retrieval techniques, that were not designed to be used with crash reports, outperform other techniques which are specifically designed for the task of crash bucketing at realistic industrial scales. This research indicates that automated crash bucketing still has a lot of room for improvement, especially in terms of identifier tokenization.

CCS Concepts
• Information systems → Near-duplicate and plagiarism detection; • Software and its engineering → Software testing and debugging; Maintaining software;

Keywords

1. INTRODUCTION
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR’16, May 14-15 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4186-8/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901766

Ada is a senior software engineer at Lovelace Inc., a large software development company. Lovelace has just shipped the latest version of their software to hundreds of thousands of users. A short while later, as Ada is transitioning her team to other projects, she gets a call from the quality-assurance team (QA) saying that the software she just shipped has a crashing bug affecting two-thirds of all users. Worse yet, Ada and her team can’t replicate the crash. What would really be helpful is if every time that crash was encountered by a user, Lovelace would automatically receive a crash report [1], with some context information about what machine encountered the crash, and a stack trace [1] from each thread. Developers consider stack traces to be an indispensable tool for debugging crashed programs—a crash report with even one stack trace will help fix the bug significantly faster than if there were had no stack traces available at all [2].

Luckily for Ada, Lovelace Inc. has gone through the monumental effort of setting up an automated crash reporting system, much like Mozilla’s Crash Error Reports [3], Microsoft’s WER [4], or Apple’s Crash Reporter [5]. Despite the cost associated with setting up such a system, Ada and her team find the reports it provides are invaluable for collecting telemetric crash data [6].

Unfortunately, for an organization as large as Lovelace Inc., with so many users, even a few small bugs can result in an unfathomable amount of crash reports. As an example, in the first week of 2016 alone, Mozilla received 2 189 786 crash reports, or about 217 crashes every minute on average.1 How many of crash reports are actually relevant to the bug Ada is trying to fix?

The sheer amount of crash reports present in Lovelace’s crash reporting system is simply too much for one developer, or even a team of developers, to deal with by hand. Even if Ada spent only one second evaluating a single crash report, she would still only be able to address 1/3 of Lovelace’s crash reports received during one day of work. Obviously, an automated system is needed to associate related crash reports together, relevant to this one bug, neatly in one place. All Ada would have to do is to select a few stack traces from this crash bucket [4], and get on with debugging her

1https://crash-stats.mozilla.com/api/SuperSearch/?date=\%3d2016-01-01&date=\%3d2016-01-08 The total number of crashes will slowly increase over time and then eventually drop to zero due to Mozilla’s data collection and retention policies.
application. Since this hypothetical bucket has all crash stack traces caused by the same bug, Ada could analyze any number of stack traces and pinpoint exactly where the fault is and how to fix it.

The questions that this paper seeks to answer are:

| RQ1: | What are effective, industrial-scale methods of crash report bucketing? |
| RQ2: | How can these methods be tuned to increase precision or recall? |

This paper will evaluate existing techniques relevant to crash report bucketing, and propose a new technique that attempts to handle this fire hose of crash reports with industrially relevant upper bounds ($O(\log n)$ per report, where $n$ is number of crash reports). In order to validate new techniques some of the many techniques described in the literature are evaluated and compared in this paper. The results of the evaluation shows that techniques based on the standard information retrieval statistic, term frequency × inverse document frequency (tf-idf), do better than others, despite the fact these techniques discard information about what is on the top of the stack and the order of the frames on the stack.

1.1 Contributions

This paper presents PARTYCRASHER, a technique that buckets crash reports. It extends the work done by Lerch and Mezini [14] to the field of crash report deduplication and show that despite its simplicity, it is quite effective. This paper contributes:

1. a criterion for industrial-scale crash report deduplication techniques;
2. replication of some existing methods of deduplication (such as Wang et al. [13] and Lerch and Mezini [14]) and evaluations of these methods on open source crash reports, providing evidence of how well each technique performs at crash report bucketing;
3. implementation of these methods in an open source crash bucketing framework;
4. evaluation based on the automated crashes collected by the Ubuntu project’s Apport tool, the only such evaluation at the time of writing;
5. a bug report deduplication method that outperforms other methods when contextual information is included along with the stack trace.

1.2 What makes a crash bucketing technique useful for industrial scale crash reports?

The volume, velocity, variety, and veracity (uncertainty) of crash reports makes crash report bucketing a big-data problem. Any solution needs to address concerns of big-data systems especially if it is to provide developers and stakeholders with value [19]. Algorithms that run in $O(n^2)$ are unfeasible for the increasingly large amount of crash reports that need to be bucketed. Therefore, an absolute upper-bound of $O(n \log n)$ is chosen for evaluated algorithms.

The methods evaluated in this paper were methods found in the literature, or methods that the authors felt possibly had promise. Methods that were evaluated in this paper were restricted to those that met the following criteria. The criteria were chosen to match the industrial scenario as described in the introduction.

1. Each method must scale to industrial-scale crash report deduplication requirements. Therefore, it must run in $O(n \log n)$ total time. Equivalently, each new, incoming crash must be able to be assigned a bucket in $O(\log n)$ time or better.
2. No method may delay the bucketing of an incoming crash report significantly, so that up-to-date near-real-time crash reports, summaries, and statistics are available to developers at all times. This requires the method to be online.
3. No method may require developer intervention once it is in operation, or require developers to manually categorize crashes into buckets. This requires the method to be unsupervised.
4. No method may require knowledge of the eventual total number of buckets or any of their properties beforehand. Each method must be able to increase the number of buckets only when crashes associated with new faults arrive due to changes in the software system for which crash reports are being collected. This requires the method to be non-stationary.

Several deduplication methods are evaluated in this paper. They can be categorized into two major categories. First, several methods based on selecting pre-defined parts of a stack to generate a signature were evaluated. The simplest of these methods is the 1Frame method, that selects the name of the function on top of the stack as a signature. All crashes that have identical signatures are then assigned to a single bucket, identified by the signature used to create it.

Similarly, signature methods 2Frame and 3Frame concatenate the names of the two or three functions on top of the stack to produce a signature. 1Addr selects the address of the function on top of the stack to generate a signature rather than the function name. 1File selects the name of the source file in which the function on top of the stack is defined to generate a signature, and 1Mod selects either the name of the file or the name of the library, depending on which is available. Figure 1 shows an example stack trace and how the various signatures are extracted from it using these methods. All of the signature-based methods, as implemented, run in $O(n \log n)$ total time or $O(\log n)$ amortized time.

The second category of methods are those based on tf-idf [22] and inverted indices, as implemented by the off-the-shelf information-retrieval software ElasticSearch 1.6 [23]. tf-idf is a way to normalize a token based on both its occurrence in a particular document (in our case, crash reports), and inversely proportional to its appearance in all documents. This means that common tokens that appear frequently in nearly all crash reports have little discriminative power compared to tokens that appear quite frequently in a small set of crash reports.

1.3 Background

Of course, the idea of crash bucketing is not new; Mozilla’s system performs bucketing [7, 6], as does WER [4]. Many approaches make the assumption that two crash reports are similar if their stack traces are similar. Consequently, researchers [8, 9, 10, 11, 4, 7, 12, 13, 14, 15] have proposed various methods of finding similar stack traces, crash report
similarity, crash report deduplication, and crash report bucketing. In order to motivate the evaluation and design choices it is necessary to look at what already has been proposed.

Empirical evidence suggests that a function responsible for crash is often at or near the top of the crash stack trace [8, 2, 15]. As such, many bucketing heuristics employ higher weighting for grouping functions near the top of the stack [10, 4, 13]. Many of these methods are similar to or extensions of the 1Frame method, that assumes that the function name on the top of the stack is the most (or only) important piece of information for crash bucketing. However, at least one study refutes the effectiveness of truncating the stack trace [14]. The most influential discriminative factors seem to be function name [14] and module name [11, 4].

Lerch and Mezini [14] did not directly address crash report bucketing; they addressed bug report deduplication through stack trace similarity. They deduplicated bug reports that included stack traces by comparing the traces with tf-idf, which is usually applied to natural language text. Although crash bucketing was implicit in this approach to bug-report deduplication, the authors did not compare this technique against the other crash report deduplication techniques. Unlike the signature-based methods, tf-idf-based methods do not consider the order that frames appear on the stack. A function at the top of the stack is treated identically to a function at the bottom of the stack.

This paper applies and evaluates Lerch and Mezini [14]'s method of bug report deduplication to crash report deduplication, both excluding contextual data from the crash report as suggested by Lerch and Mezini [14] and including it. These methods are listed in the evaluation section as the Lerch method and the LerchC method, respectively. The automated crash reporting tools collected contextual data at the same time as the crash stack trace. This paper also evaluates variants of the Lerch and LerchC methods. Space, SpaceC, Camel, and CamelC were created for this evaluation based on tokenization techniques described by [23] and by including or excluding contextual information available in the crash reports. The variants replace the tokenization pattern used in Lerch and LerchC with a different tokenization pattern. The name specifies the kind of tokenization: Space splits on whitespace only; Camel splits intelligently on CamelCasedComponents. If the name is followed by a C, the evaluation included the entire context of the stack trace along with the stack trace itself. Figure 1 shows how each method tokenizes a sample stack frame.

Liu and Han [9] grouped crashes together if they suggest the same fault location. The fault locations were found using some stack similarity metric, and found that the most discriminative power is in the top-most stack frames—i.e., the functions that are closer to the crash point.

14 Methods Not Appearing In This Report

Mozilla’s deduplication technique, as the time of writing, as it is implemented in Socorro [24] requires a large number of hand-written regular expressions to select, ignore, skip, or summarize various parts of the crash report. These must be maintained over time by Mozilla developers and volunteers in order to stay relevant to crashes as versions of Firefox are released. This technique typically uses one to three of the frames of the stack and likely has similar performance to 1Frame, 2Frame, and 3Frame. Furthermore, the techniques employed by Mozilla are extremely specific to their major product, Firefox, while the evaluation dataset contains crashes from 616 other systems.

In 2005, Brodie et al. [8] presented an approach that normalizes the call stack to remove non-discriminative functions as well as flattening recursive functions, and compares stacks using weighted edit distance. Since pairwise stack matching would be infeasible on large data sets—having a minimum worst case run-time of $O(n^2)$—they index a hash of the top $k$ function names at the top of the stack and use a B+Tree look-up data structure. Several approaches since have used some stack similarity metric, and found that the most discriminative power is in the top-most stack frames—i.e., the functions that are closer to the crash point.

Figure 1: An example stack trace (top), its various signatures (middle), and various tokenizations of the top line of the trace (bottom).
unfeasible, as they are not easy to incorporate into already existing software, and often incur pairwise comparisons to bucket regardless of instrumentation cost. Methods that already assume buckets such as Kim et al. [18] and Wu et al. [15] are disregarded as well.

Modani et al. [10] propose several algorithms. The first algorithm employs edit distance, requiring $O(n^2)$ total time. The second and third algorithms are similar, employing longest common subsequences and longest common prefixes, respectively. The longest common subsequence problem is, in general, NP-hard in the number of sequences (corresponding to crashes for the purposes of this evaluation). The longest common prefix algorithm can be implemented sufficiently efficiently for the purposes of this evaluation, but was not evaluated here because it first subdivides buckets as the 1Frame algorithm, that already creates too many buckets. Thus no Modani et al. [10] comparison algorithms were used.

Bartz et al. [11] also used edit distance on the stack trace, but a weighted variant with weights learned from training data. Consequently, they were able to consider other data in the crash report aside from the stack trace. The weights learned suggested some interesting findings: substituting a module in a call stack resulted in a much higher distance; as well, the call stack edit distance was found to be the highest-weighted factor, despite the consideration of other crash report data, confirming the intuition in the literature of the stack trace’s importance.

The methods based on edit distance—viz., Brodie et al. [8], Modani et al. [10], Bartz et al. [11]—are disqualified due to their requirement of pairwise comparisons between stack traces, with an upper-bound of $O(n^2)$.

Schröter et al. [2] empirically studied developers’ use of stack traces in debugging and found that bugs are more likely to be fixed in the top 10 frames of their respective crash stack trace, further confirming the surprising significance of the top-k stack frames in crash report bucketing, which is also corroborated more recently by Wu et al. [15].

The method described in Dhaliwal et al. [7] is not included in the evaluation because it first subdivides buckets produced by the 1Frame deduplication method, and requires $O(\lvert B\rvert^2)$ total time to run, where $\lvert B\rvert$ is the number of buckets. Its use of the 1Frame method already produces a factor of 1.67 times too many buckets. Despite the optimization in Dhaliwal et al. [7] that attempts to avoid $O(n^2)$ behaviour, it has $O(\lvert B\rvert^2)$ behaviour. Since the number of buckets increases over time, though at a slower rate, this method will eventually become computationally unfeasible if old data is not discarded.

Dang et al. [12] created a model that places more weight on stack frames closer to the top of the stack, and favours stacks whose matched functions are similarly spaced from each other. This technique suffers from a proposed $O(n^3)$ clustering algorithm.

Wang et al. [13] created three “rules” for finding correlations between crash stack traces: rule 1 correlates the method signature found in one stack trace to be contained in the other; rule 2 correlates stack traces if the source file name on the top frame of the stacks are the same; rule 3 finds closed ordered subsets of file names that are found in the stack traces. It weighs these subsets by the relative frequency of finding this ordered subset in a bucket. The only method from Wang et al. [13] directly evaluated in this paper is the method of comparing file names at the top of the stack, as 1Frame.

Thus, there are many approaches for bucketing crash reports and crash report similarity, but some are less realistic or industrially applicable than others. Any new work in the field must attempt to compare itself against some of the prior techniques such as Lerch and Mezini [14].

2. METHODOLOGY

First, the requirements for an industrial-scale automated crash deduplication system were characterized by looking at systems that are currently in use. Then, a variety of methods from the existing literature were evaluated for applicability to the task of automated crash report deduplication. Several methods that met the requirements were selected. A general purpose Python framework in which any of the selected deduplication methods could be supported and evaluated was developed, and then used to evaluate all of the methods by simulating the process of automated crash reports arriving over time. Additionally, a dataset that could be used as a gold set to judge the performance of such methods was obtained. The dataset was then filtered to include only crash reports that had been deduplicated by human developers and volunteers.

Various approaches of automatic crash report categorization (the exact problem that Ada is tasked with solving) is simulated. First, a crash report arrives with no information other than what was gathered by the automated reporting mechanisms on the user’s machine. This report might include a description written by the user of what they were
2.1 Mining Crash Reports

The first step in the evaluation procedure is mining of crash reports from Ubuntu’s bug repository, Launchpad [20]. This was done using a modified version of Bicho [21], a software repository mining tool. Over the course of one month, Bicho was able to retrieve 126,609 issues from Launchpad, including 80,478 stack traces in 44,465 issues. Some issues contain more than one stack trace. For issues that contained more than one stack trace, the first stack trace posted to that issue was selected, yielding 44,465 issues with crash reports and stack traces. The first stack trace is selected because it is the one that arrives with the automated crash report, generated by the user’s machine.

Ubuntu crash reports were used for the evaluation because they are automatically generated and submitted but many of them have been manually deduplicated by Ubuntu developers and volunteers. Other data sources, such as Mozilla’s Crash Reports, have already been deduplicated by Mozilla’s own automated system.

Next, the issues were put into groups based on whether they were marked as duplicates of another issue, resulting in 30,664 groups of issues. These groups are referred to as “issue buckets” for the remainder of the paper, to prevent conflating with groups of crash reports, that will be referred to as “crash buckets.” This dataset is available.

2.1.1 Stack Trace Extraction

Each issue and stack trace obtained from Ubuntu is formatted as plain text, as shown in Figure 3. They were then parsed into JSON-formatted data with individual fields for each item, such as address, function name, and which library the function came from. Unfortunately, this formatting is not always consistent and may be unusable. For example, some stack traces contain unintelligible binary data in place of the function name. This could be caused by memory corruption when the stack trace was captured. 2,216 crash reports and stack traces were thrown out because their formatting could not be parsed, leaving 41,708 crash reports with stack traces.

2.2 Crash Bucket Brigade

Issues were then filtered to only those that had been deduplicated by Ubuntu developers and other volunteers, yielding 15,293 issues with 15,293 stack traces in 3,824 issue buckets. These crash reports were submitted to Launchpad by the Apport tool. They were collected over a one month period. Because Launchpad places restrictions on how often the Launchpad API can be used to request data, and each crash report required multiple requests, it required over 20 seconds to download each issue. The crash reports used in the evaluation span 617 different source packages, each of which represents a software system. The only commonalities between them are that they are all written in C, C++, or other languages that compile to binaries debuggable by a C debugger, and that they are installed and used on Ubuntu. The most frequently reported software system is Gnome, which has 2,154 crash reports with stack traces. This dataset is large, comprehensive and covers a wide variety of projects.

2.3 Deciding when a Crash is not Like the Others

For methods based on Lerch and Mezini, there is a threshold value, T, that determines how often, and when, an incoming crash report is assigned to a new bucket. A specific value for T was not described by Lerch and Mezini, so a range of different values from 1.0 to 10.0 were evaluated. Higher values of T will cause the algorithm to create new buckets more often.

The threshold value applies to the score produced by the Lucene search engine inside ElasticSearch 1.6 [23]. Details of this tf–idf based scoring method are described within the ElasticSearch documentation.6 The scoring algorithm is based on tf–idf, but contains a few minor adjustments intended to make scores returned from different queries more comparable.

2.4 Implementation

The complete implementation of the evaluation presented in this paper is available in the open-source software PARTY-CRASHER.7 The implementation includes every deduplication method we claimed to evaluate above, a general-purpose

http://github.com/orezpraw/Bicho/

http://archive.org/details/bugkets-2016-01-30

http://launchpad.net/apport

https://github.com/orezpraw/Bicho/

https://archive.org/details/bugkets-2016-01-30
deduplication framework, the programs used to mine and filter the data used for the evaluation, the programs that produced the evaluation results, the raw evaluation results, and the scripts used to plot them.

2.5 Evaluation Metrics

Two families of evaluation metrics were used. These are the BCubed precision, recall, and F1-score, and the purity, inverse purity, and F1-score. Both are suitable for characterizing the performance of online non-stationary clustering algorithms by comparing the clusters that evolve over time to clusters created by hand. A comparison of BCubed and purity, along with several other metrics, and an argument for the advantages of BCubed over purity is provided in Amigó et al. [25]. The mathematical formulae for both metrics can be found in Amigó et al. [25]. However, purity also has an advantage over BCubed: specifically that it does not require $O(n^2)$ total time to compute whereas BCubed does.

If a method has a high BCubed precision, this means that there would be less chance of a developer finding unrelated crashes in the same bucket. This is important to prevent crashes caused by two unrelated bugs from sharing a bucket, possibly causing one bug to go unnoticed since usually a developer would not examine all of the crashes in a single bucket.

If a method has a high BCubed recall, this means that there would be less chance of all the crashes caused by a single bug to become separated into multiple buckets. Reducing the scattering of a single bug across multiple buckets is important as scattering interferes with statistics about frequently experienced bugs.

In contrast, purity and inverse purity focus on finding the bucket in the experimental results that most closely matches the bucket in the gold set. Then the overlap between the two closest matching buckets is used to compute the purity and inverse purity metrics, with high purity indicating that most of the items in a bucket produced by one of the methods evaluated are also in the matching bucket in the gold set. High recall indicates that most of the items in a bucket from the gold set are found in the matching bucket produced by the method being evaluated.

The purity method does not, however, completely reflect the goals of the evaluation. Purity and inverse purity do not capture anything besides the overlap between the two buckets that overlap the most. So, if a method creates a bucket that is 51% composed of crashes from a single bug, the other 49% doesn’t matter. That 49% could come from a different bug, or 200 different bugs, but the purity would be the same value. It is included in this evaluation for completeness, since it was used by Dang et al. [12].

Both metrics can be combined into F-scores. In this evaluation, F1-scores were used, placing equal weight on precision and recall (or purity and inverse purity.)

BCubed and purity can be used with the gold set, hand-made buckets that are available from Ubuntu’s Launchpad [20] bug tracking system. Ubuntu developers and volunteers have manually marked many of the bugs in their bug tracker as duplicates. Furthermore, many of the bugs in the bug tracker are automatically filed by Ubuntu’s automated crash reporting system, Apport. This evaluation uses only bugs that were both automatically filed by Apport and manually marked as duplicates of at least one other bug. The dataset is biased to the distribution of crashes that are bucketed, which might be different than crashes that are not. Conversely, this prevents the evaluation dataset from containing any crashes that have not yet evaluated by an Ubuntu developer or volunteer.

3. RESULTS

After extracting crash reports from Launchpad, and implementing various crash report bucketing algorithms, the performance of these algorithms on the Launchpad gold set was evaluated. Evaluation is multifaceted as in most information retrieval studies since the importance of either precision or recall are tuneable.

3.1 BCubed and Purity

Evaluation of the performance of bucketing algorithms is performed with BCubed and purity metrics. Figure 4 shows the performance of a variety of deduplication methods evaluated against the entire gold set of deduplicated crash reports. The 1File and 1Addr methods have the most precision, while Lerch has the most recall. F1-score is dominated by CamelC and Lerch. As in the results of Lerch and Mezini [14], using only the stacks outperforms using the stack plus its metadata and contextual information in terms of F1-score. For the CamelC, Lerch, and LerchC simulations, a threshold of $T = 4.0$ was used.

Amigó et al. [25] observed differences in BCubed and purity metrics. Their observation was tested empirically by the evaluation. In Figure 4, BCubed and purity showed similar results. The best and worst methods in terms of BCubed precision are the same as the best and worst methods in terms of purity; the same holds true for BCubed recall and inverse purity, and BCubed F1-score and purity F1-score. However, some of the methods with intermediate performance are much closer together in purity F1-score than they are in BCubed F1-score.

Figure 4 also shows that in general, if a method has a higher precision or purity, it also has a lower recall and inverse purity. For example, 3Frame has a higher precision than 2Frame, having a higher precision than 1Frame, but 1Frame has a higher recall than 2Frame and 3Frame.

The CamelC crash bucketing method employs: tf–idf; a tokenizer that attempts to break up identifiers such as variable names into their component words; and the entire context of the crash report including all fields reported in addition to the stack. It outperforms other bucketing methods evaluated.

3.2 Bucketing Effectiveness

Figure 5 shows the number of buckets created by a variety of deduplication methods. The number of issue buckets extracted from the Ubuntu Launchpad gold set is plotted as the line labelled Ubuntu. The method that created a number of buckets most similar to the number mined from the Ubuntu Launchpad gold set was LerchC. The Lerch and LerchC simulations, a threshold of $T = 4.0$ was used.

Figure 6 shows the performance of the Lerch method when used with a variety of different new-bucket thresholds, $T$. Figure 7 shows the number of buckets created by the same method with those same thresholds. Since Lerch and Mezini [14] did not specify what threshold they used, this evaluation explored a range of thresholds. It can be seen from the plots that the relative performance of $T$ thresholds,
Figure 4: BCubed (top) and Purity-metric (bottom) scores for various methods of crash report deduplication.
more desirable to tune the value of $T$ by using direct developer feedback rather than the technique employed here, comparing against an existing dataset. Instead of using data, one could ask developers if they had seen too many crashes caused by unrelated bugs in a single bucket recently. If they had, then $T$ should be increased. Or, $T$ should be decreased if developers see multiple buckets that seemed to be focused on crashes caused by the same bug.

3.3 Tokenization

Threshold is not the only way that a trade-off between precision and recall can be made. A variety of methods were tested that use the ElasticSearch/Lucene tf–idf-based search from Lerch and Mezini [14], but do not follow their tokenization strategy. The performance of several tokenization strategies is shown in Figure 9. As in other cases, the methods with high precision had low recall, and the methods with high recall had low precision. All methods shown in Figure 9 used a threshold of $T = 4.0$.

The Space method is obtained by replacing the tokenization strategy in Lerch with one that splits words on whitespace only, such that it does not discard any tokens regardless of how short they are, and does not lowercase every letter in the input. The Space method performs worse than Lerch. However, when both stack traces and context are used, the Space method, performance improves slightly. This is the opposite behaviour of Lerch. Adding context (LerchC) causes performance to decrease slightly. A third tokenization strategy, Camel was evaluated. Camel attempts to break words that are written in CamelCase into their component words, using a method provided in the ElasticSearch documentation. This strategy had the worst performance of the three, until it was used with context included, called CamelC. The addition of context allowed CamelC to outperform every other method evaluated in this paper.

The worst-performing tokenization evaluated, 1Addr, was also the method that produced the largest number of buckets. However, tuning methods to match the number of buckets in the gold set without concern for performance did not result in higher performance. Lerch with $T = 3.0$ and Space with $T = 4.0$ were not the best-performing threshold or method, but both produced almost the same number of buckets as the gold set.

3.4 Runtime Performance

The current implementation of PartyCrasher requires only 45 minutes to bucket and ingest 15 293 crashes, using the slowest algorithm, CamelC, on a Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz machine with 32GiB of RAM and a Hitachi HDS723020BLE640 7200 RPM hard drive. Performance depends mainly on disk throughput, latency and RAM available for caching; ElasticSearch recommends using only solid-state drives. This works out to 335 crashes per minute, meeting the performance goal of 217 crashes per minute based on crash-stats from Mozilla.

4. DISCUSSION

4.1 Threats to Validity

Results are dependent on the gold set—a manual classification of crash report by Ubuntu volunteers. The results

https://github.com/elastic/elasticsearch/blob/1.6/docs/reference/analysis/analyzers/pattern-analyzer.asciidoc
Figure 6: BCubed scores for the Lerch method of crash report deduplication at various new-bucket thresholds $T$.

Figure 7: Number of buckets created as a function of number of crashes seen for the Lerch method of crash report deduplication at various new-bucket thresholds $T$. The line labelled Ubuntu indicates the number of groups crashes that were marked as duplicates of each other by Ubuntu developers or volunteers.

Figure 8: Precision/Recall plot showing the tradeoff between BCubed precision and recall as the new-bucket threshold $T$ is adjusted. BCubed $F_1$-score is also listed in the plot.
may be biased due to the exclusive use of known duplicate crashes; the known and classified duplicates may not be representative of all crash reports. If any of these methods with tunable parameters are deployed, the parameters should be tuned based on feedback from people working with the crash buckets, not just the gold set.

Since the evaluation only used data from open source software, it is unknown if our results are applicable to closed-source domains. Only stacks that originate from C and C++ projects have been evaluated; it is possible that other languages, compilers, and their runtimes have different characteristics in how they form stack traces. However, these results are corroborated by studies that examined Java exclusively [13, 14].

4.2 Future Work

The results presented indicate that improvements could be made to tf–idf-based-crash deduplication methods. For instance, 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.

The tokenization techniques evaluated in this paper are extremely primitive. They are merely regular expressions that break up words based on certain types of characters such as spaces, symbols, uppercase letters, lowercase letters and numbers. Advanced tokenization techniques, such as the ones found in Guerrouj et al. [27] and Hill et al. [28], would likely outperform the basic techniques that have been evaluated in this paper.

It would be valuable to measure the effectiveness of using the buckets produced by the CamelC technique as input to other methods, such as those that perform bug triaging [26] and crash localization [15].

5. CONCLUSION

The results in this paper indicate that off-the-shelf tf–idf-based information retrieval tools can bucket crash reports in a completely unsupervised, large-scale setting when compared to a variety of other previously proposed algorithms. Based on these results, a developer, such as Ada, should choose a tf–idf-based crash deduplication method with tokenization that fits their dataset, and intermediate new-bucket threshold. They should update this threshold based on feedback from developers, volunteers, or employees that work with the stack traces directly. A tf–idf approach that used the entire crash report and stack trace, tokenized using camel-case had the best F1-score on the Ubuntu Launchpad crash reports used in this work. In addition, there is a lot of room for improvements to these techniques. This conclusion is surprising in light of the fact that the tf–idf-based techniques evaluated disregard information that is often considered to be essential to stack traces, such as the order of the frames in the stack.

Finally the research questions can be answered:

RQ1: tf–idf-based methods are effective, industrial-scale methods of crash report bucketing.
RQ2: New-bucket thresholds and tokenization strategies can be tuned to increase precision and recall.

Acknowledgements

The authors would like to thank the Mozilla foundation, especially Robert Helmer, Adrian Gaudebert, Peter Bengtsson and Chris Lonnen for their help, and for making Mozilla’s massive collection of stack reports open and publicly available. Additionally, the authors would like to thank the Ubuntu project and all of its developers who manually deduplicated bug reports that were submitted with crash reports. Funding for this research was provided by MITACS Accelerate with BioWare™, an Electronic Arts Inc. studio.
6. REFERENCES


