Empirical Software Engineering manuscript No.
(will be inserted by the editor)

TraceSim: An Alignment Method for Computing Stack Trace Similarity

Irving Muller Rodrigues · Aleksandr Khvorov · Daniel Aloise · Roman Vasiliev · Dmitrij Koznov · Eraldo Rezende Fernandes · George Chernishev · Dmitry Luciv · Nikita Povarov ·

Received: date / Accepted: date

Abstract Software systems can automatically submit crash reports to a repository for investigation when program failures occur. A significant portion of these crash reports are duplicate, i.e., they are caused by the same software issue. Therefore, if the volume of submitted reports is very large, automatic grouping of duplicate crash reports can significantly ease and speed up analysis of software failures. This task is known as crash report deduplication. Given a huge volume of incoming reports, increasing quality of deduplication is an important task. The majority of studies address it via information retrieval or sequence matching methods based on the similarity of stack traces from two crash reports. While information retrieval methods disregard the position of a frame in a stack trace, the existing works based on sequence matching algorithms do not fully consider subroutine global frequency and unmatched

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson, Ciena, and EffciOS for funding this project. Moreover, this research was enabled in part by the support provided by WestGrid (https://www.westgrid.ca/) and Compute Canada (www.computecanada.ca).

Irving Muller Rodrigues, Daniel Aloise
Polytechnique Montreal, Montreal, Canada
E-mail: {irving.muller-rodrigues, daniel.aloise}@polymtl.ca

Aleksandr Khvorov
HSE University, JetBrains, St Petersburg, Russia
E-mail: aleksandr.khvorov@jetbrains.com

Roman Vasiliev, Nikita Povarov
JetBrains, St Petersburg, Russia,
E-mail: {roman.vasiliev, nikita.povarov}@jetbrains.com

George Chernishev, Dmitrij Koznov, Dmitry Luciv
Saint-Petersburg State University, Saint-Petersburg, Russia
E-mail: {g.chernyshev, d.koznov, d.lutsiv}@spbu.ru

Eraldo Rezende Fernandes
FACOM – UFMS, Campo Grande, Brazil
E-mail: eraldo@facom.ufms.br

frames. Besides, due to data distribution differences among software projects, parameters that are learned using machine learning algorithms are necessary to provide more flexibility to the methods. In this paper, we propose TraceSim – an approach for crash report deduplication which combines TF-IDF, optimum global alignment, and machine learning (ML) in a novel way. Moreover, we propose a new evaluation methodology for this task that is more comprehensive and robust than previously used evaluation approaches. TraceSim significantly outperforms seven baselines and state-of-the-art methods in the majority of the scenarios. It is the only approach that achieves competitive results on all datasets regarding all considered metrics. Moreover, we conduct an extensive ablation study that demonstrates the importance of each TraceSim's element to its final performance and robustness. Finally, we provide the source code for all considered methods and evaluation methodology as well as the created datasets.

Keywords Duplicate Crash Report, Crash Report Deduplication, Duplicate Crash Report Detection, Automatic Crash Reporting, Stack Trace

1 Introduction

Many software products are nowadays equipped with automated crash reporting systems such as Apport[1], Mozilla Socorro[2], and CrashPad[3]. These systems detect software crashes, collect data related to user environment, system state and execution information (Ahmed et al, 2014), and group such data into a so-called *crash report*. While automated crash reporting systems reduce the dependence on users to collect relevant information about failures, they drastically increase the number of crash reports. For instance, according to Campbell et al (2016), Mozilla Firefox received around 2.2 million crash reports in the first week of 2016. A significant portion of these crash reports were duplicates, i.e., multiple reports related to the same software bug. For example, we found that 72% of the reports of the IntelliJ Platform (a JetBrains product family) were duplicates.

In software projects, duplicate crash reports are grouped into clusters called *buckets*. This grouping helps to prioritize the bug fixing, provides supplemental information about a failure, and reduces the effort required to fix a bug (Glerum et al, 2009; Dhaliwal et al, 2011). However, due to the massive volume of crash reports submitted daily, it is unfeasible to manually allocate new reports to buckets. For instance, considering that 13,000 crash reports were submitted per hour for Mozilla Firefox (Campbell et al, 2016) and, that a "superhuman" could review one report per second, a triager would take around 3.6 hours to identify the buckets of these new reports. Hence, it is vital for large software projects to automatically assign crash reports to buckets. This task is known as *duplicate*

[1] https://wiki.ubuntu.com/Apport
[2] https://crash-stats.mozilla.com/
[3] https://goto.google.com/crash/root

crash report detection, *crash report bucketing* or *crash report deduplication* (Campbell et al, 2016; Dang et al, 2012).

During the program lifetime, a stack (named *call stack*) keeps track of active subroutines. We consider a subroutine active if it is under execution or waiting for the completion of other subroutines. Call stacks are composed of frames: data structures that store information on a single active subroutine (such as its return address and arguments). These frames are stored following the LIFO (last in, first out) principle, i.e., the frames related to the last executed subroutines are on the top of the stack. The *stack trace* is hence a snapshot of the call stack in memory which is captured and presented to the user when a system crashes.

In Figure 1, we illustrate a crash report. This example presents the details about the system and environment in **lines 1–7**. This information is variable and depends on the application. Moreover, reports may include user descriptions of bugs and how they could be reproduced. In Figure 1, **lines 9–43** represent a stack trace. It encompasses valuable information for developers to understand and fix an error (Schroter et al, 2010).

```
 1  Date: 2016-01-20T22:11:48.834Z
 2  Product: XXXXXXXXXXXX
 3  Version: 144.3143
 4  Action: null
 5  OS: Mac OS X
 6  Java: Oracle Corporation 1.8.0_40-release
 7  Message: new child is an ancestor
 8
 9  java.lang.IllegalArgumentException: new child is an ancestor
10    at javax.swing.tree.DefaultMutableTreeNode.insert(DefaultMutableTreeNode.java:179)
11    at javax.swing.tree.DefaultMutableTreeNode.add(DefaultMutableTreeNode.java:411)
12    at com.openapi.application.impl.ApplicationImpl$8.run(ApplicationImpl.java:374)
    .....
41    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
42    at java.lang.Thread.run(Thread.java:745)
43    at org.ide.PooledThreadExecutor$2$1.run ....
```

Fig. 1: Crash report example

The majority of previous studies addresses crash report deduplication mainly by measuring similarity between the stack traces of two crash reports. Lerch and Mezini (2013), Campbell et al (2016), and Sabor et al (2017) use traditional information retrieval techniques to compute this similarity. These works propose to encode stack traces as vectors whose dimensions are related to subroutine names (fully-qualified names of functions) and values are calculated using *Term Frequency – Inverse Document Frequency* (TF-IDF) (Manning and Schütze, 1999). One key drawback of information retrieval methods is that they ignore the order of the subroutines in a stack trace. Other studies (Dang et al, 2012; Modani et al, 2007; Brodie et al, 2005; Bartz et al, 2008) explicitly consider the sequence of function calls within stack traces and employ variants of sequence matching algorithms (such as edit distance, longest common subsequence and

optimal global alignment) to measure the similarity between two stack traces. Brodie et al (2005) were the first to use a sequence matching algorithm to compute the similarity between two stack traces. Moreover, in the matching algorithm, they considered two pieces of information to weight the importance of a subroutine in a given stack trace: its position in the stack trace and its frequency in a large database. The rationale behind this is twofold: (i) bugs are more likely to be related to subroutines in the top positions of the stack trace (Schroter et al, 2010); and (ii) rare subroutines are more relevant than frequent ones, similarly to TF-IDF. Later, Dang et al (2012) proposed PDM, a method that also considers subroutine position when comparing two stack traces by means of a sequence matching algorithm. PDM, different from Brodie et al (2005), includes a machine learning (ML) algorithm to learn the parameters that control the impact of frame position in the matching algorithm. In that way, PDM can adapt to different data distributions from varied software projects or even to temporal shifts in the same project. One neglected aspect in the current literature is the proper consideration of subroutines that exist in only one of the two stack traces under comparison. We call these subroutines *unmatched* as opposed to matched subroutines that are present in both stack traces. In Brodie et al (2005), for instance, position and frequency are considered only for matched subroutines. For unmatched ones, the similarity score is given by a negative constant value. In Dang et al (2012), unmatched subroutines are ignored when computing the similarity score. However, these missing subroutines may be important to estimate the similarity (or, in this case, the dissimilarity) between stack traces, especially the ones that are rare and lie in the top positions.

In this work, we propose *TraceSim*, the first technique that structurally combines TF-IDF, optimum global alignment, and machine learning for crash report deduplication. To compute the similarity between two stack traces, TraceSim finds the optimum global alignment between them by means of the Needleman-Wunsch (NW) algorithm. Differently from previous approaches, we leverage the flexibility of this global alignment algorithm to consider all subroutines, either matched or unmatched, to compute the similarity score between two stack traces. Moreover, for all subroutines, TraceSim considers both their frequency in a large database (TF-IDF) and their position in the stack traces. Additionally, TraceSim employs a ML algorithm to learn parameters that regulate the influence of all these elements within the NW algorithm. Hence, TraceSim can be viewed as a generalization of the methods proposed by both Brodie et al (2005) and Dang et al (2012).

Inspired by works on bug deduplication, we also propose a new evaluation methodology for crash report deduplication. Our comprehensive methodology considers different aspects of the problem, such as the system capacity to separate non-duplicates from duplicates, the accuracy of report assignment to buckets, and the system performance for ranking. By means of this methodology, we compare TraceSim to state-of-the-art and baseline systems using four datasets from open-source projects (Ubuntu, Eclipse, Netbeans and Gnome) and one industrial dataset (JetBrains). Additionally, a detailed ablation study

is performed to assess distinct elements of TraceSim, namely TF-IDF, global aligment and ML.

The main contributions of this paper are summarized as follows:

1. We propose TraceSim, a novel method for crash report deduplication that combines TF-IDF, global alignment, and machine learning. We experimentally demonstrate that each one of these methodological choices significantly contributes to TraceSim's performance and robustness.
2. We report on the most comprehensive experimental evaluation which includes many previous methods in the literature and assesses different aspects of these methods. The experiments are performed on five distinct projects involving distinct programming languages and characteristics. TraceSim significantly outperforms the competitive methods in the majority of the scenarios, and it is the only method that consistently performs well in all scenarios.
3. We provide our full evaluation framework[4] which comprises: datasets generated from open source projects, implementations of all considered methods, and implementation of the proposed evaluation methodology. The provided framework is crucial for reproducibility, so that future works can easily compare new methods with Tracesim and other relevant methods from the literature within a comprehensive and unified framework.

The remainder of this paper is organized as follows. Section 2 describes the proposed method. Section 3 presents the existing techniques that address crash report deduplication. Section 4 presents the proposed evaluation methodology and the experimental setup. In Section 5, we experimentally compare TraceSim to competitive techniques and report on several ablation studies. Section 6 discusses the possible threats to the validity of our work. Finally, concluding remarks are given in Section 7.

2 TraceSim

A common assumption in the literature is that failures caused by the same bug are represented by similar system executions. Since a stack trace can capture the state of a system execution right before a crash, crash deduplication usually proceeds by mainly comparing the similarity between stack traces. In order to compute this similarity, many studies employ sequence matching algorithms so that they can measure the overlapping between two stack traces keeping track of the order of the compared frames.

A classic sequence matching method is the Needleman–Wunsch (NW) algorithm (Needleman and Wunsch, 1970), which finds the optimal global alignment between two sequences. A global alignment consists in aligning the elements of two sequences end-to-end. In Figure 2, we illustrate a global alignment of two stack traces ($stack_1$ and $stack_2$). Filled rectangles represent

[4] Temporary link: `https://drive.google.com/file/d/1ZXj1DBqKDiVU3GZ4iU_fVzD0IxiLlmCU`. A Github repository link will be added to the final version.

frames and empty ones symbolize gaps: special structures that allow to shift the position of an element in the alignment. As can be observed, frames can be lined up to gaps, where each gap represents that a specific subroutine is missing in that position of the stack trace. We call a *match* the alignment of identical frames (e.g., the frames `Maps.difference` in $stack_1$ and $stack_2$). In contrast, we name the alignment of two different frames a *mismatch* (e.g., the frames `ValidatorPage.performOk` and `BuilderPage.schedCleaner`). In this work, a mismatch is considered as equivalent to two gap alignments since it is unlikely that two different subroutines possess interchangeable functionality (Brodie et al, 2005). For instance, `ValidatorPage.performOk` and `BuilderPage.schedCleaner` are clearly different and, therefore, it is appropriate to align these frames to gaps.

Two stack traces can be aligned in multiple ways. The optimal global alignment problem is formulated as a maximization problem such that each possible frame alignment (match, mismatch, and gap) has an assigned value. Thus, the optimum solution for the problem corresponds to the global alignment for which the sum of the match values minus the sum of the mismatch and gap values is maximum. A common scheme is to define the values for matches,

$stack_1$	$stack_2$
	HashMap.putMapEntries
util.HashMap.<init>	HashMap.<init>
Maps.difference	Maps.difference
OptCfgBlock.getPreferenceChanges	OptCfgBlock.getPreferenceChanges
OptCfgBlock.processChanges	
ValidatorPage.performOk	BuilderPage.schedCleaner
PreferenceDialog.run	PreferenceDialog.run
SafeRunner.run	SafeRunner.run
JFaceUtil.run	JFaceUtil.run
SafeRunnable.run	SafeRunnable.run
PreferenceDialog.okPressed	PreferenceDialog.okPressed
PreferenceDialog.buttonPressed	PreferenceDialog.buttonPressed
Dialog.widgetSelected	Dialog.widgetSelected
TypedListener.handleEvent	TypedListener.handleEvent

Fig. 2: Best Global Alignment Example: 12 matches, 1 mismatches, and 2 gaps. Matches, mismatches and gaps are represented by blue, red, and yellow, respectively.

mismatches and gap alignments as constants, such that a specific alignment is always associated to the same value. In order to leverage the peculiar characteristics of stack traces, we employ a scheme that computes the alignment values in a more effective way for the crash deduplication task. In this scheme, weights are assigned to each frame. These weights estimate the frame importance to discriminate two stack traces and are used to compute match, mismatch

and gap values. Our hypothesis is that some frames are more relevant than others for comparison and, hence, their correct or wrong alignment should have higher impact on the computed similarity score. The NW algorithm is more suitable for our proposed scheme than other classic sequence matching algorithms, e.g. longest common subsequence (LCS) and Levenshtein distance (also known as edit distance). In the LCS, the similarity is only affected by the matches since mismatches are not allowed and gap values are zero. Hence, the incorrect alignment of important frames is not considered for measuring the similarity. Conversely, the matches of relevant and irrelevant frames are not distinguishable in the edit distance because the match values are always zero. The NW algorithm is the method that allows us to fully consider the weights of all frames to measure the similarity of stack traces.

In the remainder of this section, we describe *TraceSim*, our proposed method for crash report deduplication. This method computes the similarity of two stack traces s^q and s^c from a new query report q and a candidate report c, respectively. Each stack trace is represented as a sequence of frames, i.e., $s^q = (s^q_1, s^q_2, \ldots, s^q_{|s^q|})1$ and $s^c = (s^c_1, s^c_2, \ldots, s^c_{|s^c|})$, where s^q_1 and s^c_1 are the frames at the top of the stack traces, and $|s^q|$ and $|s^c|$ are the number of frames in s^q and s^c, respectively. Following the majority of the studies, we only consider two frames as equal when their subroutine names are exactly the same. In order to compare s^q and s^c, TraceSim first assigns a weight to each frame of these two stack traces. Then, by means of the NW algorithm, our method finds the optimal maximum global alignment between s^q and s^c by considering the match, mismatch, and gap values proportional to the frame weights. The NW algorithm runs in $O(|s^q| \cdot |s^c|)$ time. Finally, the score of the optimal alignment is normalized by a technique based on the *Jaccard index* (Chierichetti et al, 2010). The normalization is fundamental for an effective comparison of alignment scores since it adjusts the scores by the frame weights of the stack traces.

2.1 Frame Weight Computation

Bugs are more frequently located in the top frames of stack traces (Schroter et al, 2010). Hence, it is natural to consider that frames near the top are more relevant for crash deduplication than the bottom ones. Nonetheless, frames, including those in the top positions, could be associated to subroutines that are common through the database, e.g, subroutines that are related to logging, thread pooling, error-handling and entry points. These subroutines poorly accommodate the discrimination of similar stack traces since they appear in multiple crashes caused by different errors. Therefore, we consider that a frame's importance for crash report deduplication (*frame weight*) depends on two factors: its position in the stack trace (*local weight*) and its frequency in the database (*global weight*).

The local weight of the i-th frame in a stack trace s is computed as follows:

$$\mathrm{lw}(s_i) = \frac{1}{i^{\alpha}}, \tag{1}$$

where $\alpha \in \mathbb{R}_{>0}$ is a parameter that controls the function smoothness. Equation (1) assigns larger local weight values to frames located at the top of a stack trace.

The global weight is defined based on a well-known information retrieval technique: Term Frequency – Inverse Document Frequency. In our context, the term frequency (TF) is always equal to 1 since the global weight is computed for a frame in a specific position within the stack trace. This is due to the fact that the alignment algorithm is intrinsically dependent on the order of the frames and not only on their frequencies, like in the original TF-IDF. Therefore, given a crash report database S, the inverse document frequency (IDF) of a frame s_i is simply defined as:

$$\mathrm{IDF}(s_i) = \frac{|S|}{\mathrm{df}(s_i)},$$

where $|S|$ is the total number of stack traces in the database S and $\mathrm{df}(s_i)$ is the document frequency of the subroutine s_i, i.e., the number of stack traces that contain a subroutine s_i among the set S of stack traces. Hence, the global weight of the i-th frame in a stack trace s is computed as follows:

$$\mathrm{gw}(s_i) = e^{-\frac{\beta}{\mathrm{IDF}(s_i)}}, \tag{2}$$

where $\beta \in \mathbb{R}_{>0}$ is a parameter that controls the function smoothness. The rarer a subroutine is, the larger are the values computed by Equation (2).

Finally, the weight of a frame s_i is defined as:

$$\mathrm{w}(s_i) = \mathrm{lw}(s_i) \times \mathrm{gw}(s_i). \tag{3}$$

2.2 Optimal Global Alignment

Being based on the Needleman-Wunsch algorithm, TraceSim applies dynamic programming to find the optimal global alignment between stack traces s^q and s^c as follows. Let us define a matrix M in which $M_{i,j}$ is the optimal alignment score between the subsequences $s^q_1, s^q_2, \ldots, s^q_i$ and $s^c_1, s^c_2, \ldots, s^c_j$. The matrix M is iteratively computed using a bottom-up strategy:

$$M_{i,j} = \max \begin{cases} M_{i-1,j} + \mathrm{gap}(s^q_i) \\ M_{i,j-1} + \mathrm{gap}(s^c_j) \\ M_{i-1,j-1} + F(s^q_i, s^c_j) \end{cases}, \tag{4}$$

where $F(s^q_i, s^c_j)$ is given by:

$$F(s^q_i, s^c_j) = \begin{cases} \mathrm{mismatch}(s^q_i, s^c_j), & \text{if } s^q_i \neq s^c_j \\ \mathrm{match}(s^q_i, s^c_j), & \text{otherwise} \end{cases}. \tag{5}$$

The *match*, *mismatch* and *gap* values are calculated by the functions match(·), mismatch(·), and gap(·), respectively. These values are proportional to the frames' weights and represent their discriminative power.

The first and second lines in Equation (4) are associated with frames aligned to gaps. Since this is an incorrect alignment of only one frame, the gap value for a frame s' is computed as:

$$\text{gap}(s') = -\,\text{w}(s'). \tag{6}$$

The first line in Equation (5) denotes a mismatch between s_i^q and s_j^c (the two frames are different). Since gap alignments are preferable to mismatches for crash deduplication, the mismatch value is equivalent to lining up the two frames to gaps. This is expressed as:

$$\text{mismatch}(s_i^q, s_j^c) = -\,\text{w}(s_i^q) - \text{w}(s_j^c). \tag{7}$$

The second line in Equation (5) denotes a match between s_i^q and s_j^c. Inspired by Dang et al (2012), the function match(·) in TraceSim is defined as:

$$\text{match}(s_i^q, s_j^c) = \max(\text{w}(s_i^q), \text{w}(s_j^c)) \times \text{diff}(s_i^q, s_j^c). \tag{8}$$

We assume that stack traces emerging from the same error contain subroutines in the same region of the stack trace. Therefore, in Equation (8), the maximum weight between the two matched frames is normalized by the diff(·) function that measures the alignment offset of two frames as follows:

$$\text{diff}(s_i^q, s_j^c) = e^{-\gamma|i-j|},$$

where $\gamma \in \mathbb{R}_{>0}$ is a parameter that controls the exponential function smoothness. Thus, we penalize matches in which the positions of the matched frames are discrepant.

Finally, the score of the best global alignment between s^q and s^c is defined as:

$$\text{align}(s^q, s^c) = M_{|s^q|,|s^c|}. \tag{9}$$

2.3 Normalization

Figure 3 illustrates three stack traces ($stack_3$, $stack_4$ and $stack_5$) and their respective frame weights. By applying the algorithm described in Section 2.2, we obtain $\text{align}(stack_3, stack_4) = -0.66$ and $\text{align}(stack_3, stack_5) = -0.67$.

According to these alignment scores, $stack_3$ is slightly more similar to $stack_4$ than to $stack_5$. However, this is counter-intuitive since all frames in $stack_3$ and $stack_4$ are different while $stack_3$ and $stack_5$ share four subroutines in the same positions. This problem occurs because all frames in $stack_4$ have a low weight and, consequently, the resulting gaps and mismatches do not present a significant impact to the maximum score. As such, it is ineffective to compare alignment scores because they depend on the frame weights of the compared stack traces.

$stack_3$

A	B	C	D	E
0.01	0.2	0.4	0.01	0.01

$stack_4$

F	H	I
0.01	0.01	0.01

$stack_5$

A	B	G	D	E
0.01	0.2	0.5	0.01	0.01

Fig. 3: Normalization report example

Moreover, in order to help users to interpret similarity scores, it is desired to limit scores within a range. Note that according to (9), the alignment score is unbounded, i.e., its value might vary between $-\infty$ to $+\infty$. Besides, there exists an asymmetry in the algorithm since match$(\cdot)$ depends on the maximum of the frame weights whereas mismatch$(\cdot)$ is defined by the sum of the weights. Due to these two characteristics, it is challenging to normalize the alignment scores using canonical normalization, e.g. via min-max scaling. Based on the assumption that the proportion of shared subroutines between two stack traces is a good indicator for the deduplication, we propose a normalization inspired by the weighted Jaccard index (Deza and Deza, 2016), which computes the similarity between two documents Z and Y as:

$$\text{jaccard}(Z, Y) = \frac{\sum_{k=1}^{|T|} \min(z_k, y_k)}{\sum_{k=1}^{|T|} \max(z_k, y_k)}, \tag{10}$$

where T is the set of unique terms in the dataset (called *vocabulary*); and $z, y \in \mathbb{R}^{|T|}$ are vector representations of Z and Y, respectively. Each dimension of z and y corresponds to a specific term in the vocabulary T. This representation is called vector space model (VSM) (Manning and Schütze, 1999).

In crash deduplication, stack traces can be cast as documents and subroutines in the frames as vocabulary terms. Let us consider V as a vocabulary of subroutines. Thus, a stack trace $s = (s_1, \ldots, s_{|s|})$ can be represented as a vector $x \in \mathbb{R}^{|V|}$ whose k-th dimension is given by:

$$x_k = \sum_{i=1}^{|s|} \text{w}(s_i) \times \mathbb{1}[s_i = t_k], \tag{11}$$

where t_k is the k-th term in V and $\mathbb{1}[s_i = t_k]$ returns 1 when the subroutine in a frame s_i is equal to t_k, and 0 otherwise. In summary, we assign zero to the dimensions of x whose associated subroutines in V do not appear in a stack trace. In the opposite case, the dimension value is the sum of all frame weights $\text{w}(s_i)$ of a specific subroutine t_k.

Given the corresponding vectors x^q and x^c for the stack traces s^q and s^c, respectively, we normalize the score of the maximum global alignment by:

$$\text{sim}(s^q, s^c) = \frac{\text{align}(s^q, s^c)}{\sum_i^{|V|} \max(x_i^q, x_i^c)}. \tag{12}$$

Thus, $\text{sim}(s^q, s^c)$ belongs to the interval $[-1, 1]$.

2.4 Machine learning

The three parameters of TraceSim (α, β, and γ) are tuned via a machine learning technique: *Tree-structured Parzen Estimator* (TPE) – a Bayesian hyperparameter optimizer (Bergstra et al, 2013b). TPE finds parameter values that maximize the sum of two metrics on a given tuning set. Then, such parameters are used on a subsequent validation set for final evaluation. In Section 4.4, we describe the training and evaluation processes, including the optimized metrics. In Sections 4.2 and 4.3, we introduce four additional parameters, also tuned using TPE, that control some preprocessing procedures.

3 Related Work

The optimal global alignment as well as the longest common subsequence, the edit distance, and the longest prefix match are well-known sequence matching problems. These problems have been extensively studied over the decades and have been applied to many different domains. In the literature, many studies have addressed crash report deduplication as one of these sequence matching problems.

Brodie et al (2005) proposed a variant of the Needleman-Wunsh (NW) algorithm to compare two stack traces. Similarly to TraceSim, its match value depends on three factors: the position and document frequency of the matched frame in the new report and the alignment offset of the two matched frames. However, unlike TraceSim, the method of Brodie et al (2005) does not contain parameters that control the influence of the frame position and document frequency on the match value. Thus, it cannot adapt to the software project particularities, e.g., the relevance of frame positions for crash deduplication may vary among applications. Moreover, its gap and mismatching values are constant, i.e., they do not depend on frame positions and subroutine document frequencies. Therefore, the optimal alignment score is equally affected by incorrectly matching 1) rare subroutines located at the top, and 2) frequent ones located at the bottom. Finally, the alignment scores are not regularized by the stack trace length and the document frequencies.

Two studies – Bartz et al (2008) and Dhaliwal et al (2011) – proposed techniques based on edit distance for crash report deduplication. Edit distance measures the dissimilarity between two sequences as the minimum number of edit operations (insertions, removals and substitutions) required to convert one

of the sequences into another (see e.g. (Miller et al, 2009)). It was shown by Sellers (1974) that edit distance and optimal global alignment are equivalent problems. Bartz et al (2008) designed a logistic regression to calculate the probability of crash reports being duplicate. As features, this method uses the edit distance between two stack traces and categorical data comparisons (event type, process name and exception code). For computing edit distance, the substitution cost depends on the modules, offsets and subroutines of the frames. Besides that, insertion and deletion penalties have different values when a new group (a subsequence of frames with the same module) is created or removed. This method assumes that module and offset information are always present in C/C++ stack traces, which is not necessarily true. Dhaliwal et al (2011) proposed to organize crash reports with a two-level grouping scheme. First, they created a first-level group (coarse granularity) that contains reports with the same frame in the top position. After that, they reorganized the reports in the first level into subgroups (fine granularity). These subgroups are generated based on the edit distance between the reports. The drawback of these two studies is that they ignore two important pieces of frame information: position and document frequency.

Modani et al (2007) reported that prefix match achieves better precision and recall values than the edit distance and the technique proposed by Brodie et al (2005). Prefix match considers the similarity of two stack traces as the length of the longest common prefix between the stack traces normalized by the size of the longest stack trace. One drawback of Prefix match is that small differences in the top and middle positions can highly affect the computed similarity.

Dang et al (2012) applied an agglomerative hierarchical clustering technique to cluster crash reports. To compute stack trace similarities, they proposed a method called position dependent model (PDM) that finds the optimal common subsequence of two stack traces. PDM employs an algorithm similar to the Needleman-Wunsh algorithm but for which the gap and mismatch values are zero, i.e., they do not affect the final solution score. Like TraceSim, the match value is computed using the position of the nearest frame to the top and the alignment offset of the matched frames. However, PDM does not consider the document frequency of the frames to compute the similarity score. Therefore, frequent subroutines in the top positions of the stack can highly affect the similarity, even though these subroutines may occur in the top positions of many unrelated stack traces. Moreover, the similarity score is not affected by neither mismatches nor gaps.

Another group of studies proposed techniques based on information retrieval. Lerch and Mezini (2013) proposed to use the TF-IDF technique (implemented by Lucene[5]) to calculate the similarity of stack traces. Campbell et al (2016) compared the TF-IDF method (implemented by ElasticSearch[6]) with signature-based methods. According to them, these methods are appropriate for industrial

[5] `https://lucene.apache.org/`

[6] `https://www.elastic.co/elasticsearch/`

projects since such environments require a search complexity of $O(n \log n)$ where n is the number of reports. In their work, two crash reports were considered duplicate when their similarity score was greater than a defined threshold. The authors found TF-IDF to be superior to other techniques. Besides, the authors proposed a new tokenization that tokenizes camel-cased texts, achieving better cluster metric values by using all data from crash reports.

Sabor et al (2017) proposed DURFEX – a new technique for crash report deduplication. In order to reduce sparsity, DURFEX employs package names instead of fully-qualified method signatures. Besides that, the n-grams of the package names are generated to keep the temporal order of the frames. After preprocessing, DURFEX converts the stack traces to vectors using TF-IDF. The similarity between two reports is then given by the linear combination of the features generated from categorical data comparisons with the cosine similarity of stack trace vectors.

Moroo et al (2017) proposed a re-ranking scheme to combine sequence matching methods with information retrieval. First, they use the TF-IDF method to generate a ranked list of the most similar reports to a query. Then, PDM is employed to calculate a new similarity score for the top-k reports. Finally, the top-k reports are reordered based on the weighted harmonic mean of TF-IDF and PDM. This combination of techniques is limited since a subroutine's document frequency and positions are considered independently for the comparison. TraceSim can compare the frame orders of stack traces using this supplementary information.

Three studies propose methods for crash report deduplication focusing on report and bucket comparison. Kim et al (2011) developed a method called CrashGraph that represents both stack traces and buckets as graphs. The nodes of a graph represent the subroutines, and the edges link nodes whose subroutines are adjacent within the stack traces. The similarity between a stack trace and a bucket is computed as the percentage of edges shared between their graph representations. Koopaei and Hamou-Lhadj (2015) proposed CrashAutomata: a method that generates n-grams for each stack trace, and then prunes the n-grams whose frequencies exceed a given threshold. An automata is generated for each bucket based on the extracted n-grams of the stack traces. CrashGraph and CrashAutomata can be negatively affected by bucket heterogeneity and they ignore the document frequencies of the subroutines. Ebrahimi et al (2019) trained a Hidden Markov Model (HMM) for each bucket of crash reports. These HMM models are used to detect whether a crash report belongs to a bucket. This method is not scalable since an HMM model has to be trained for each bucket. Moreover, it cannot be applied in software projects whose buckets can contain only one report.

Unlike the existing matching algorithms, TraceSim combines the position and document frequency of frames to compute weights, providing an estimation of the frame's importance to crash deduplication. As supported by the results in Section 5, we believe that this scheme improves the method's capability to distinguish relevant frames from irrelevant ones and, consequently, it helps the method to better adjust the similarity score based on the correct (or

wrong) matching of frames. Moreover, unlike information retrieval techniques, TraceSim leverages the document frequency of the subroutines without losing track of the frame order.

4 Experimental Setup

In this section, we present the main components of our experimental setup: the datasets used in the experiments, preprocessing steps, strategies to compare reports with multiple stack traces, our evaluation methodology, and competing baseline methods. The datasets from open-source applications and the developed code are available online[7].

4.1 Datasets

Open data sources that contain crash reports are scarce and the few available are unfit for investigating crash report deduplication. For instance, Mozilla maintains a repository[8] of crash reports related to their products, but bucket assignment is performed automatically by their own system which hinders an accurate evaluation. Therefore, in the literature, one popular alternative for this problem is to mine bug tracking systems (BTS) of open-source applications in order to extract crash reports that include stack traces. Some of these BTSs include manually assigned buckets.

We use four datasets generated from BTS data of open-source projects. Campbell et al (2016) created a crash report dataset from bug reports in the Ubuntu bug repository[9] comprising issues from 617 different software systems for Ubuntu that are compatible with the C debugger. We have generated three other datasets from bug reports of three popular BTSs: Eclipse[10], Netbeans[11] and Gnome[12]. We only considered reports submitted before January 1st 2020. NetBeans and Eclipse are well-known integrated development environments (IDEs) developed in Java, while Gnome's BTS keeps track of bugs from 648 software projects (applications, libraries, bindings, among others) developed for the GNOME desktop environment. We extracted stack traces from the description field and attached files of bug reports. To better imitate real crash reports, we remove the attachments uploaded at most ten minutes after the report creation. This timespan has shown to be suitable for removing files that were uploaded after the bug report inspection by the triaging team. The parser developed by Lerch and Mezini (2013) was employed to extract stack traces

[7] `https://drive.google.com/file/d/1ZXj1DBqKDiVU3GZ4iU_fVzD0IxiLlmCU`
[8] `https://crash-stats.mozilla.org/`
[9] `https://bugs.launchpad.net/`
[10] `https://bugs.eclipse.org/bugs/`
[11] `https://bz.apache.org/netbeans/`
[12] `https://bugzilla.gnome.org/`

from Eclipse and NetBeans BTSs, while the Parse::StackTrace[13] module was used to extract stack traces from Gnome BTS.

At some point after a new report is submitted, a user of a bug tracking system analyzes and assigns it to either an existing (duplicate report) or a new bucket (new bug). Thus, there is a time gap between the report submission and the triage assessment. In the meantime, a recently submitted report can be incorrectly labeled. We believe that the span of one year substantially reduces label instability. Therefore, in the Eclipse dataset, we only keep reports created before 2019. NetBeans started to gradually migrate their reports to another BTS in the middle of 2017. Thus, we only consider reports submitted until 2016 for Netbeans. Finally, for Gnome's BTS, we have found a significant reduction in the number of submitted reports with stack traces after 2011 (only $\sim 2.2\%$ of all reports in Gnome were created between 2012 and 2018). The cause of this decrease is unknown, thus we have decided to remove reports submitted during this period to avoid undesirable bias.

Besides open-source projects, we evaluate our method using data from the JetBrains crash report processing system Exception Analyzer. This system handles reports from various products of the IntelliJ Platform product family, which includes IntelliJ Idea, PyCharm, Kotlin Plugin and others. Its products have a large user base, and their maintainers receive several hundreds of crash reports per day. If a product from the IntelliJ Platform crashes, then Exception Analyzer receives the generated report. Newly arrived crash reports are fed to the classification algorithm, which either assigns the report to an existing issue (which means that the report corresponds to an existing bug) or leaves it without treatment. In the latter case, the report status is unclassified and it is considered that a new bug is encountered. Next, unclassified crash reports are grouped together using a clustering algorithm. These groups are then manually inspected by an on-duty QA engineer who is selected among the developers of IntelliJ Platform every day. The QA engineer can accept the generated issue candidate and thus, create a new issue, or decline it. The clustering algorithm can create meaningless issues by combining crash reports belonging to different bugs. In this case, the QA engineer can manually move reports belonging to the faulty issue to other issues or leave them untouched. Moreover QA engineer can analyze and move reports from one issue to another, a more suitable one, doing that not only for new issues or new reports, but also for existing ones. QA engineer provides continued activity of supporting stack trace database in a consistent state. The latter approach may be reasonable since new, but similar crash reports can arrive later and then automatic clustering can create a new issue correctly. Finally, new issues are passed to the developers of IntelliJ Platform for fixing. Both classifying and clustering algorithms rely on stack trace comparison.

The statistics of Ubuntu, Netbeans, Eclipse, Gnome, and JetBrains datasets are presented in Table 1. For Jetbrains, we show the total number of crash reports (including the automatically classified ones) and the number of manually

[13] `https://metacpan.org/pod/Parse::StackTrace`

labeled reports inside parenthesis. For the other datasets, we use manually labeled data only. It is important to highlight that Jetbrains, Nebtbeans, and Eclipse are composed of Java stack traces whereas Ubuntu and Gnome consist of C/C++ stack traces.

Table 1: Statistics of datasets. The number of manually labeled reports are shown inside parenthesis for JetBrains. In the datasets of open-source projects, only manually labeled data are used.

Dataset	Period	# Duplicates	# Reports	# Buckets
Ubuntu	2007/05/25 - 2015/10/18	11,468	15,293	3,825
Eclipse	2001/10/11 - 2018/12/31	8,332	55,968	47,636
Netbeans	1998/09/25 - 2016/12/31	13,703	65,417	51,714
Gnome	1998/01/02 - 2011/12/31	117,216	218,160	100,944
Jetbrains	2018/08/09 - 2020/05/20	880,476 (6,516)	925,233 (51,273)	44,757

4.2 Preprocessing

The structure of stack traces depends on the programming language. An example of a stack trace in both C/C++ and Java is depicted in Figure 4.

Fig. 4: Stack trace example

As shown in this figure, Java stack traces are typically composed of exception class name and message, subroutine names (fully-qualified name of the method) and location in source code. A subroutine source consists of the file and the

line where a subroutine was paused. Stack traces in C/C++ may present a wide variety of information about each frame but their content depends on the debugger and system libraries. For instance, in Figure 4, the C/C++ stack trace contains frame positions, frame pointer addresses, subroutine names, arguments, and sources. For both programming languages, *we only extract the subroutine names and the position of the frames*. All remaining information is ignored for crash report deduplication.

We found some subroutine names inconsistencies in C/C++ stack traces. In order to correct them, subroutine names were preprocessed using the following steps. First, since in some cases the parser could not accurately separate arguments from subroutine names, we had to search for these inconsistencies and remove them from subroutine names manually. After that, we stripped the pattern __GI__ and underscore symbols (_) from the beginnings of names since these prefixes are likely inserted by the debugger or compiler. Therefore, following these steps, for instance, __GI___libc_free (mem=0x3) and __libc_free are transformed to libc_free.

When a debug package of a software system is not installed on a machine, the stack trace may contain frames for which information about the subroutine call is missing. In these frames, subroutines are represented as ?? in C/C++ and HIDDEN.HIDDEN in Java. We refer to these subroutines as *unknown subroutines*. In our experiments, we test two different strategies to handle such subroutines: the first approach considers them as equivalent for stack trace comparison, while the second treats them as different. Although the first strategy prevents the wrong comparison of different subroutines, the second one can detect patterns of subsequences with unknown subroutines.

Removing recursion is an important preprocessing step (Brodie et al, 2005; Modani et al, 2007). We test two different recursion removal algorithms. The first algorithm, proposed by Brodie et al (2005), removes subsequent frames with the same subroutine names. The second one, developed by Modani et al (2007), strips all frames that occurred between two similar frames of the same subroutine. Finally, to remove uninformative functions, we employed the unsupervised algorithm created by Modani et al (2007). In this method, a frame is considered uninformative when the document frequency percentage of its subroutine name is higher than a threshold. Consecutive uninformative frames in the top and bottom positions are removed from the stack traces.

4.3 Multiple Stack Traces

Crash reports may include multiple stack traces, mainly due to multi-processing and multi-threading systems. When such systems crash, each process or thread usually generates a specific stack trace. Since the information of which thread-/process that caused the crash may be unknown, all stack traces are considered for deduplication. Another cause is related to the data characteristics. In Netbeans and Eclipse BTSs, bug reports can include multiple stack traces within their description and attached files. According to Schroter et al (2010), these

extra stack traces provide additional information about an issue. Thus, we keep all stack traces found in a report. Finally, stack traces from a nested exception were considered as different stack traces since some of them are related to process/thread executions and the extraction method proposed by Lerch and Mezini (2013) incorrectly considers a significant amount of nested exceptions to be single stack traces. In Table 2, we present the number of reports with multiple stack traces.

Table 2: Number of reports with multiple stack traces (ST), total number of reports, and the ratio of these two quantities for each dataset.

BTS	**# Reports w/ Multiple ST**	**# Reports**	**Ratio**
Ubuntu	5	15,293	0.03%
Eclipse	13,641	55,968	24.37%
Netbeans	40,147	65,417	61.37%
Gnome	174,841	218,160	80.14%

In order to compute the similarity between two crash reports q and c that contain multiple stack traces, we compute the similarity of all possible pairs of stack traces, in which one member of the pair belongs to the query crash report q and the other comes from the candidate crash report c. Thus, a similarity matrix $S \in \mathbb{R}^{m \times n}$ is created where $S_{i,j}$ is the similarity between the i-th and j-th stack traces in q and c, respectively. We then assess six different strategies to reduce this matrix to a real number. The first strategy performs the reduction as follows:

$$\text{max\_stg}(S) = \max_{1 \leq i \leq m, 1 \leq j \leq n} S_{i,j}. \tag{13}$$

Basically, this strategy returns the highest value in the similarity matrix.

The next five strategies perform matrix reduction by applying a maximum operation followed by a mean operation. The first one is defined as follows:

$$\text{query\_stg}(S) = \frac{1}{m} \sum_{i}^{m} \max_{1 \leq j \leq n} S_{i,j}. \tag{14}$$

query_stg$(\cdot)$ computes the average of the maximum similarity of each stack trace in the query. A similar strategy can be applied to the stack traces in the candidate:

$$\text{cand\_stg}(S) = \text{query\_stg}(S^\intercal). \tag{15}$$

Instead of considering the stack trace sources, the third strategy calculates the mean of maximum values of the shortest report (report that contains the smallest number of stack traces):

$$\text{short\_stg}(S) = \begin{cases} \text{query\_stg}(S), & \text{if } m \leq n \\ \text{cand\_stg}(S), & \text{otherwise.} \end{cases} \tag{16}$$

The opposite strategy is defined as follows:

$$\text{long\_stg}(S) = \begin{cases} \text{query\_stg}(S), & \text{if } m \geq n \\ \text{cand\_stg}(S), & \text{otherwise.} \end{cases} \tag{17}$$

Finally, the last approach is:

$$\text{avg\_stg}(S) = \frac{\text{query\_stg}(S) + \text{cand\_stg}(S)}{2}. \tag{18}$$

4.4 Proposed Evaluation Methodology

In previous literature, there is no widely adopted methodology for evaluation of crash report deduplication systems. Nevertheless, two common approaches are *ranking* (Lerch and Mezini, 2013; Sabor et al, 2017) and *binary classification* (Bartz et al, 2008; Modani et al, 2007). Both approaches have their own strengths, as well as some key limitations. In ranking approaches, a query crash report q is given and its similarity to each previously submitted report is computed. Then, a ranked list of candidate reports, sorted by decreasing similarity to q, is evaluated by means of classic ranking metrics. The higher the system ranks duplicates of q in that list, the better its performance is. However, ranking metrics are usually not defined for singleton (non-duplicate) queries. This is a key drawback of ranking methodologies because they disregard the ability of a system to filter out singleton crash reports. This is highly undesirable given the large volume of crash reports submitted to a typical crash report system. In contrast, binary classification approaches focus exactly on this filtering task. Such approaches tackle the classification problem of predicting if a query crash report is either duplicate or non-duplicate. However, they ignore one, if not the most, important aspect of crash report deduplication: identification of reports concerning the same software bug. In summary, the blind spot of binary classification approaches is covered by ranking approaches, and vice-versa.

Other popular approaches (Campbell et al, 2016; Dang et al, 2012; Moroo et al, 2017) treat crash report deduplication as a clustering problem by considering buckets of reports as clusters. The clustering metrics consider the *global solution* to measure the grouping quality, i.e., all reports are used to compute the evaluation metrics. However, in practice, software projects frequently contain an initial repository in which submitted reports are already pre-assigned to buckets. Since these previously submitted reports are considered in our evaluation methodology, we have opted to not using clustering metrics here. Instead, we have adopted metrics that are not affected by the presence of earlier reports.

Bug report deduplication is a problem related to crash report deduplication. Its input is a textual description of a software issue. The body of literature regarding this problem is larger than that for crash report deduplication. Inspired by Banerjee et al (2017), which proposed an evaluation methodology for

bug report deduplication, our evaluation methodology combines both ranking and binary classification metrics. For ranking, we use two classic metrics: Mean Average Precision (MAP) and Recall Rate@k (RR@k) (Sun et al, 2011). For binary classification, we use the classic Area Under the ROC Curve (AUC). Thus, our methodology can measure the system capacity to filter duplicate and non-duplicate reports, compute the percentage of duplicate reports correctly assigned to buckets, and evaluate ranking quality. Additionally, Rakha et al (2018), also in the context of bug report deduplication, suggest to evaluate a system on different portions of a dataset in order to better study how varies performance due to data changes. Moreover, this idea inspired us to develop our own methodology to crash report deduplication.

Query and Candidate Sets

In our methodology, a dataset of crash reports is always organized in chronological order, as this better reflects the real scenario of software engineering projects. In order to compute each metric, we first select a *query set* Q of consecutive crash reports within a given dataset. In Figure 5, we illustrate a chronologically-ordered dataset, a query set Q within it, and other key aspects of our methodology. When evaluating a system, each query report $q \in Q$ is considered as a newly submitted crash report; and reports submitted before q are considered *candidate reports*, i.e. possible duplicates of q. More specifically, for each query report $q \in Q$, we define a corresponding candidate set $C(q)$ with reports in the dataset that were submitted before q.

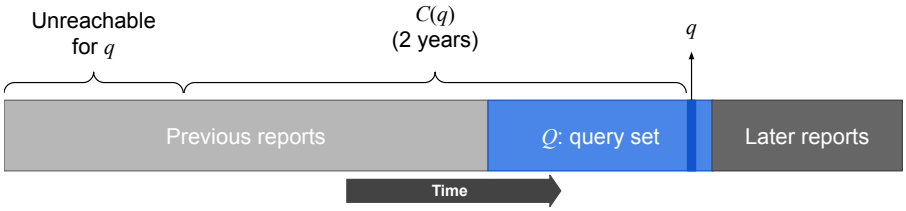


Fig. 5: Illustration of a chronologically-ordered dataset in which we select a query set Q (blue span). Given a query report $q \in Q$, its corresponding candidate set $C(q)$ is shown.

Same as bug report datasets, crash report datasets can be very large. Banerjee et al (2017) suggest to limit the candidate set $C(q)$ to a certain *time window* in the context of bug report deduplication. This way, we reduce both the computational cost of duplicate report detection and the performance degradation due to the repository growth over time. As depicted in Figure 5, we limit $C(q)$ to a time window of two years. That is, for any $q \in Q$, $C(q)$ comprises all reports in the dataset submitted at most two years before q.

Reports submitted more than two years before q are not reachable by the systems being evaluated.

In Table 3, we illustrate a dataset comprising 11 reports identified as $C1$, $C2$, ..., $C11$. The query set $Q = \{C7, C8, C9\}$ is highlighted in blue. The first step of evaluating a system for a given query set consists of computing $\text{sim}(q, c)$ for all $q \in Q$ and $c \in C(q)$. For our example, we present these values in the columns labeled as $\text{sim}(q, \cdot)$ for $q \in Q$. In these columns, a value of UR in a row c indicates that $c \notin C(q)$, i.e., report c is unreachable for query q. For example, when $q = C9$, reports $C1$, $C2$, and $C3$ are unreachable because they were submitted more than two years before $C9$. Additionally, all reports submitted after q are unreachable for q.

Table 3: Example of a dataset: query set $Q = \{C7, C8, C9\}$ (blue rows) and similarities computed by a fictitious system between each query $q \in Q$ and candidate $C(q)$. An UR label indicates that a candidate report c is *UnReachable* for a query q.

Id	**Creation Date**	**Bucket**	$\text{sim}(C7, \cdot)$	$\text{sim}(C8, \cdot)$	$\text{sim}(C9, \cdot)$
$C1$	2014/12/02	B_{C1}	0.0	0.2	UR
$C2$	2014/12/24	B_{C1}	0.0	0.5	UR
$C3$	2015/01/01	B_{C3}	0.2	0.1	UR
$C4$	2015/06/12	B_{C3}	0.7	0.0	0.8
$C5$	2016/02/22	B_{C3}	0.3	0.3	0.1
$C6$	2016/05/25	B_{C6}	0.6	0.4	0.3
$C7$	2016/05/26	B_{C6}	UR	0.0	0.2
$C8$	2016/12/01	B_{C8}	UR	UR	0.1
$C9$	2017/05/25	B_{C3}	UR	UR	UR
$C10$	2017/05/26	B_{C10}	UR	UR	UR
$C11$	2017/11/02	B_{C8}	UR	UR	UR

Limiting $C(q)$ by a time window is usually not a big issue since, in most cases, crash reports related to a bug are frequently submitted until the bug is fixed. That is, for most bugs, there is not a large gap between duplicate reports. In order to show that this is true for our datasets, we present in Table 4 the percentage of query reports that have at least one duplicate bug within a time window of two years. In the worst case (Eclipse), for less than 2.7% of all possible query reports, no previous duplicate report is reached using a window of two years.

For JetBrains, we keep the original two-month time window employed in their system. We have found that 96.60% of query reports in the Jetbrains data can reach at least one duplicate report in this time window.

Bucket-Level Metrics

In our exemplary dataset in Table 3, we include a column that indicates the bucket of each report. A bucket is identified as B_m where m corresponds to

Table 4: Percentage of query reports that reaches at least one duplicate report in a time window of two years.

Dataset	**2 years**
Ubuntu	99.47%
Eclipse	97.36%
Netbeans	98.68%
Gnome	99.50%

its master report, i.e., the first submitted (oldest) report in the bucket. For instance, bucket $\{C6, C7\}$ is denoted B_{C6}. When evaluating a system, we consider that the correct buckets in $C(q)$ are known. Thus, a system does not need to predict duplicate reports, but duplicate buckets, instead. In that way, our metrics are defined in terms of *candidate buckets* instead of candidate reports. We denote $C^B(q)$ the set of candidate buckets for a query q. This set is derived from $C(q)$, that is, $C^B(q)$ comprises buckets whose reports are in $C(q)$. For example, we have that $C^B(C7) = \{B_{C1}, B_{C3}, B_{C6}\}$. We then define the similarity $\text{sim}(q, B)$ between a query report $q \in Q$ and a bucket $B \in C^B(q)$ as the maximum similarity between q and a candidate report $c \in B$, that is:

$$\text{sim}(q, B) = \max_{c \in B} \text{sim}(q, c). \tag{19}$$

In our example, we have $\text{sim}(C7, B_{C3})$=0.7. As shown by Equation (19), all reports from B, even those outside the two-year time window, are considered to compute $\text{sim}(q, B)$. This is natural since we want to capture the full similarity between B and q.

In Table 5, we present the similarity for all pairs $q \in Q$ and $B \in C^B(q)$ for this dataset. We can observe that some buckets in the dataset are unreachable for some queries (the UR value in the table). That is the case when, for a given query, all reports of some bucket are unreachable. For instance, when $q = C9$, bucket B_{C1} is unreachable because all its reports ($C1$ and $C2$) in the dataset are unreachable for $C9$. Regarding the same query, B_{C3} is reachable since at least one of its reports is reachable for $C9$, e.g., $C_4 \in C(C9)$. Thus, all the reports in B_{C3} are considered to calculate $sim(C9, B_{C3})$ including $C3$ that is not within the time window.

Based on the similarities between queries and their candidate buckets, our methodology evaluates a method for crash report deduplication by using *ranking* and *binary classification* metrics.

Ranking Metrics

As mentioned before, ranking metrics disregard a query that corresponds to a singleton report. For the query set $Q = \{C7, C8, C9\}$ in Table 3, when $q = C8$, our methodology considers $C8$ as a singleton report since it is the first report of its bucket (master report). Although $C11$ is a duplicate of $C8$ in the dataset, $C11$ is not considered in this case since it is submitted after

Table 5: Similarity matrix $\text{sim}(q, B)$ between each query report $q \in Q$ and each bucket $B \in C^B(q)$ for the dataset in Table 3. A UR value indicates that all reports in a bucket B are unreachable for a query q. In the last column, we present the correct bucket for each query report.

Query q	**Buckets**				**Correct Bucket**
	B_{C1}	B_{C3}	B_{C6}	B_{C8}	
$C7$	0.0	0.7	0.6	UR	B_{C6}
$C8$	0.5	0.3	0.4	UR	B_{C8}
$C9$	UR	0.8	0.3	0.1	B_{C3}

$C8$. Therefore, we only consider two duplicate reports in Q ($C7$ and $C9$) to compute the ranking metrics. The set of duplicate reports within Q is denoted $Q^d \subset Q$. Given the similarities between a duplicate query $q \in Q^d$ and each of its candidate buckets $C^B(q)$, we sort this set in descending order of similarity. We define this sorted list as $L(q) = (B_1^s, B_2^s, \ldots, B_{|C^B(q)|}^s)$, where $B_i^s \in C^B(q)$. In our example, $Q^d = \{C7, C9\}$ and we have: $L(C7) = (B_{C3}, B_{C6}, B_{C1})$ and $L(C9) = (B_{C3}, B_{C6}, B_{C8})$.

The first ranking metric is MAP which is the mean of the Average Precision (AP) for all queries in Q^d:

$$\text{MAP} = \frac{\sum_{q \in Q^d} \text{AP}(q)}{|Q^d|}.$$

In our scenario, AP is very simple because, for a query $q \in Q^d$, there is only *one* relevant bucket in $C^B(q)$ that is the correct bucket for q. For a query $q \in Q^d$, AP is given by:

$$\text{AP}(q) = \frac{1}{p},$$

where p is the position of the correct bucket for q in the sorted list of candidate buckets $L(q)$. In our example, $\text{AP}(C7) = 1/2$ and $\text{AP}(C9) = 1$. An AP equal to one means that the system ranked the correct bucket in first place.

MAP is a relevant ranking metric, especially when comparing different ranking systems. However, when we consider a realistic scenario in which a manual triage of possible duplicate reports is necessary, the Recall Rate@k metric is more informative. This metric is defined as:

$$\text{RR@}k = \frac{\sum_{q \in Q^d} \mathbb{1}_k(q)}{|Q^d|},$$

where $k \geq 1$ is an integer parameter of the metric and $\mathbb{1}_k(q)$ is an indicator function whose value is one when the correct bucket for q is ranked within the first k positions in $L(q)$. This way, RR@k is the percentage of ranked lists in which the correct bucket is within the top-k positions. RR@1 is the percentage of duplicate queries for which the correct bucket is ranked first, corresponding to the accuracy of a completely autonomous system.

Due to the two-year time window, the correct bucket of a query q might not appear in $C^B(q)$ and, therefore, its position is undefined in $L(q)$. This case occurs in real scenarios and it negatively affects the performance of crash report deduplication systems. To reproduce this impact on real environments, we consider $\mathrm{AP}(q) = 0$ and $\mathbb{1}_k(q) = 0$ for each query q whose correct bucket is not in $L(q)$.

Binary Classification Metric

As discussed before, ranking metrics have a relevant limitation: they ignore non-duplicate reports in the query set. When considering a realistic scenario in which manual triage is necessary, the ability of a system to filter out non-duplicate reports is highly valuable. In the following, we explain how to cast a similarity-based system as a binary classifier that predicts if a query report is duplicate or not.

Given a query $q \in Q$ (including singletons) and the corresponding sorted list of candidate buckets $L(q) = (B_1^s, B_2^s, \ldots)$, we use the highest similarity among all candidate buckets, that is, $\mathrm{sim}(q, B_1^s)$, as a classification score. For the example in Table 5, we have the classification scores: $\mathrm{sim}(C7, B_1^s)$=0.7, $\mathrm{sim}(C8, B_1^s)$=0.5 and $\mathrm{sim}(C9, B_1^s)$=0.8. Since $\mathrm{sim}(q, \cdot) \in [-1, 1]$, we can derive a binary classifier by defining a threshold t such that q is considered duplicate if $\mathrm{sim}(q, B_1^s) \geq t$. In our evaluation, we do not need to choose t, because we use the classic Area Under the ROC Curve metric. The ROC curve is a plot of the true positive rate versus the false positive rate for every possible classification threshold. AUC summarizes the ROC curve in one meaningful number between zero and one. For example, the AUC for the query set in Table 5 is equal to one.

Parameter Tuning and Model Validation

TraceSim and other methods include some parameters that need to be tuned. In order to avoid reporting overestimated performance, we tune parameters on a query set denoted *tuning set* T and then, using the best parameters, we report final performance on a consecutive and non-overlapping query set denoted *validation set* V. Since some concept drift along time in most crash report datasets is expected, the tuning set comprises the reports immediately preceded by the reports in the validation set. In Figure 6, we depict these two sets within a chronologically-ordered dataset. In the figure, we highlight two query reports: $q^t \in T$ and $q^v \in V$. We can notice that the candidate sets $C(q^t)$ and $C(q^v)$ can overlap, but the corresponding query sets T and V do not. The time period of a validation set V is one year. The corresponding tuning set T is delimited such that $|T^d| = 250$, that is, T contains 250 duplicate reports along with all singleton reports submitted in the same period. We have found that $|T^d| = 250$ leads to good results, and that larger tuning sets did not improve overall performance.

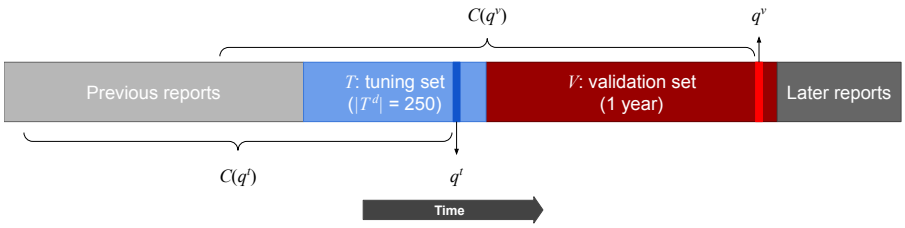


Fig. 6: Depiction of tuning and validation sets within a dataset.

As mentioned in Section 2.4, we tune parameters by means of TPE, a machine learning technique. We run TPE for 100 iterations[14] and choose the best parameters based on the sum of MAP and AUC values on the tuning set. The selected parameters are then used to compute the three considered metrics (MAP, AUC and RR@k) on the corresponding validation set. The tuned parameters are:

- the ones that control the frame weights used by the optimal global alignment algorithm: α, β and γ (Sections 2.1 and 2.2);
- the approaches to handling unknown subroutines (Section 4.2);
- the threshold to consider a subroutine as uninformative (Section 4.2);
- recursion removal algorithms (Section 4.2);
- the strategies to reduce the similarity matrix (Section 4.3).

Because of the natural concept drift in our datasets, the performance of a single method usually varies a lot from one query set to another, even within the same dataset. Thus, Rakha et al (2018) suggested to perform experimental evaluation on different portions of the dataset. Based on that suggestion, we perform our experiments as follows. Along each dataset, we randomly sample 50 validation sets. For each validation set, a corresponding tuning set is selected comprising reports submitted immediately before the validation reports. In Figure 7, we illustrate an example of five randomly selected validation sets within a dataset, along with the corresponding tuning sets. As one can observe, the selected query sets may overlap.

Unlike the datasets derived from open source projects, Jetbrains contains much more reports that were labeled by an automated system. To replicate a similar experimental setup to the production environment of Jetbrains, we consider such reports for the experiments. However, to mitigate the impact of mislabelling on the evaluation, automatically classified reports in the tuning and validation sets are disregarded for computing the ranking and binary classification metrics. That is, we do not consider these reports as queries, even though they can be in the candidate sets and they are used to compute the document frequency of the subroutines. Moreover, since the Jetbrains dataset contains much more reports than the other datasets, the time period of their validation is set to only one month and the number of iterations for TPE is

[14] We conducted a preliminary investigation to find the best number of iterations.

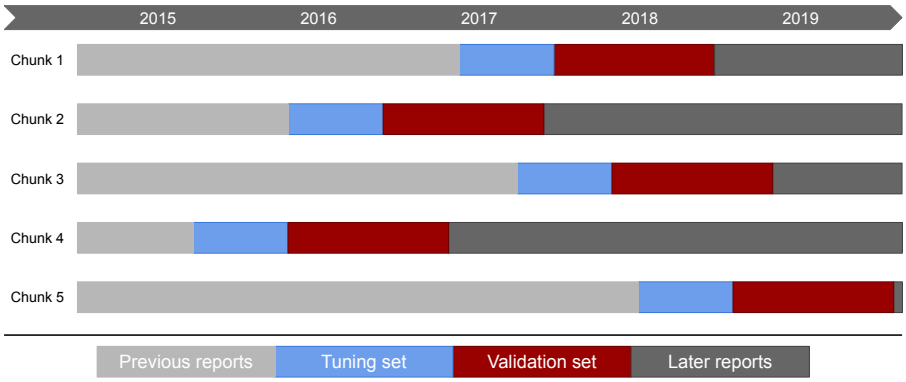


Fig. 7: Five randomly selected validation sets along with the corresponding tuning sets within the same dataset.

decreased to 50. The number of sampled validation sets and the size of the tuning set are not modified.

4.5 Competing Methods

In order to evaluate TraceSim we have selected a number of baselines. For this, competing methods from the information retrieval, string matching, and machine learning areas were selected. They have been selected due to TraceSim being a structural composition of the methods belonging to these groups.

In information retrieval category we have selected two techniques: TF-IDF and DURFEX. TF-IDF was implemented in Apache Lucene. We did not use the camel case tokenization because Campbell et al (2016) showed that it achieves poor performance when using only stack trace information. In fact, following Lerch and Mezini (2013), we treat subroutine names as single terms. DURFEX is only tested in Eclipse and Netbeans datasets since it was specifically developed for Java stack traces. Besides that, in this method, we only consider the cosine similarity of the stack trace vectors for crash report deduplication. We also compare TraceSim to five other methods: PDM, Prefix Match, the original Needleman-Wunsch (NW) algorithm, the matching algorithm proposed by Brodie et al (2005), and the reranking method designed by Moroo et al (2017). These methods are described in Section 3. Hereafter, Prefix Match is abbreviated to *PrefixM*, and we denote the techniques proposed by Brodie et al (2005) and Moroo et al (2017) as *Brodie* and *Moroo*, respectively. Finally, for the sake of fairness, all methods have access only to the positions and subroutine names of the frames in the stack traces.

We disregard some previous methods in our experiments due to different reasons. Top signature-based methods have been shown to achieve worse performances than TF-IDF (Campbell et al, 2016). The technique proposed by Bartz et al (2008) depends on features (frame offsets and module names) that

are not available in a significant portion of the stack traces in the datasets. CrashGraph (Kim et al, 2011) leverages all crash reports of a bucket to generate a bucket representation. As the majority of the works in the literature, our paper focuses on the similarity of stack traces and, therefore, CrashGraph is beyond the scope of this study. Ebrahimi et al (2019) propose a method that requires predefined buckets because an HMM is trained for each bucket. Koopaei and Hamou-Lhadj (2015) also assume a fixed number of buckets to evaluate CrashAutomata and it is uncertain how to employ this technique correctly in a scenario that does not hold such an assumption. Since our evaluation methodology considers that singletons and new buckets may be generated during evaluation, which is typical in real projects, we do not consider these two previous methods.

Regarding the competing methods with learned parameters, their original studies either do not describe the training process or use grid search to tune the hyperparameters. Since Tree-structured Parzen Estimator achieves similar or better performance than grid search (Bergstra et al, 2013b,a; Putatunda and Rama, 2018), this Bayesian optimization technique is used to tune the parameters of the competing methods for each chunk. Table 6 presents all tuned parameters for each method. For all methods, except Durfex and TF-IDF[15], we also tune the following preprocessing choices: the strategies to reduce the similarity matrix; the approaches to handling unknown subroutines; the threshold of considering a subroutine as uninformative; and the recursion removal algorithms.

Table 6: Tuned parameters for each method. For competing methods, we keep the original names.

Method	**Parameter**
TF-IDF	No learnable parameters
PrefixM	No learnable parameters
DURFEX	N-gram
NW algorithm	Match, mismatch and gap values
Brodie	Gap value
PDM	c and o
Moroo	c, o, α, and M
TraceSim	α, β and γ

5 Experimental Results

In this section, we compare TraceSim to the competing methods regarding AUC, MAP, and RR@1 on Ubuntu, Eclipse, Netbeans, Gnome, and JetBrains

[15] These strategies were designed for techniques that consider the frame order. Since these information retrieval techniques are based on the bag-of-words model, such strategies are not effective for them.

datasets. The statistics of these datasets are presented in Table 1 (Section 4.1). We also present an ablation study to assess the main components of TraceSim and investigate the effectiveness of our method of computing mismatch and gap values. As previously described, we consider 50 random validation sets for each dataset. We use the same validation sets, and the corresponding tuning sets, to evaluate all methods using the three aforementioned metrics. We then report, for each method, the distribution of each metric in the 50 validation sets using violin plots (Kampstra, 2008). These plots are produced by the standard kernel density estimate (KDE) as implemented in the `seaborn` library (Waskom, 2020). For each violin plot, we present: the estimated distribution curve; three dashed lines indicating the 25th, the 50th, and the 75th percentiles; and a white dot indicating the mean metric value.

Additionally, when comparing TraceSim to a competing method (including different versions of itself in the ablation study), we compute the difference between the performance obtained by TraceSim and the competing method in each validation set. These differences in terms of AUC, MAP, and RR@1 are denoted, respectively, ΔAUC, ΔMAP, and ΔRR@1. These values are positive whenever TraceSim outperforms a competing method on a validation set. For each competing method, we plot the distribution of the 50 differences by means of ordinary box plots. In these plots, we include a white point to indicate the mean difference between TraceSim and the competing method. Finally, following Rakha et al (2018), we apply the Wilcoxon signed-rank test (Gehan, 1965) to evaluate whether the obtained performance differences are statistically significant. The statistical hypotheses are:

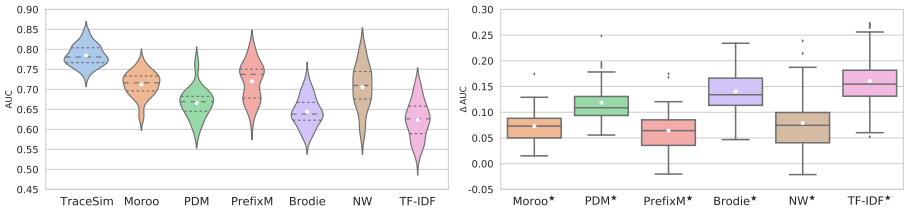
H_0: The two methods have the same performance.

H_1: The two methods have different performance.

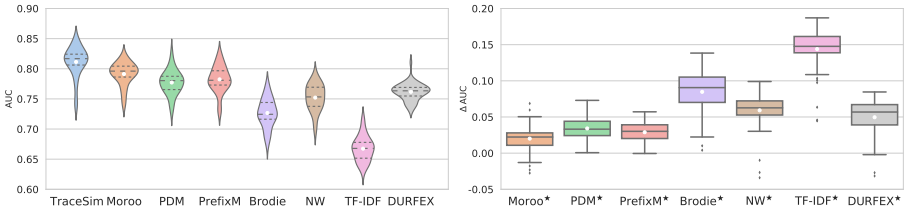
The null hypothesis (H_0) is rejected in favor of the alternative hypothesis (H_1) whenever $p < 0.01$. We indicate statistical significance by appending the symbol ★ to the name of the competing method within the corresponding box plot.

5.1 Results

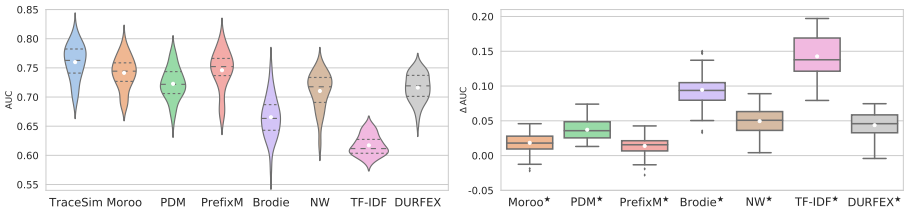
In Figure 8 (left), we present the distributions of the AUC values achieved by TraceSim and each competing method on the five considered datasets. In turn, in Figure 8 (right), we depict the performance differences between TraceSim and other methods in terms of AUC, i.e., ΔAUC. TraceSim consistently achieves competitive AUC values in all datasets. It significantly outperforms the second best technique by, on average, 6.44% in Ubuntu, 2.01% in Eclipse, and 1.39% in Netbeans. In Gnome, TraceSim substantially surpasses all methods except the NW algorithm which presents similar performance (the mean of ΔAUC is approximately zero). Finally, in the JetBrains dataset, our method yields AUC values comparable to Moroo, PDM, and PrefixM. The average of ΔAUC



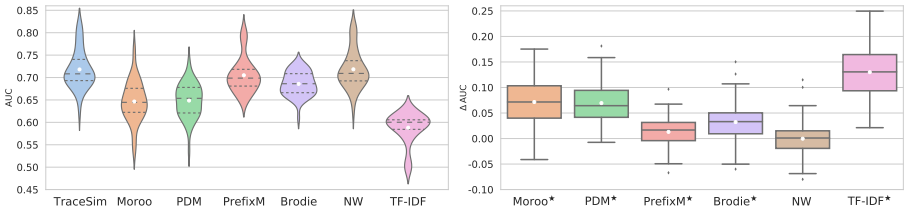
(a) Ubuntu



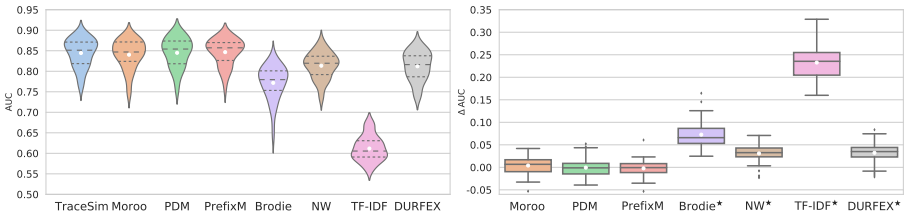
(b) Eclipse



(c) Netbeans



(d) Gnome



(e) JetBrains

★ $p < 0.01$

Fig. 8: Results regarding AUC. *Left*: distribution of AUC achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔAUC between TraceSim and each competing method.

between TraceSim and these techniques are +0.40%, -0.06%, and -0.22%, respectively. However, these differences are not statistically significant.

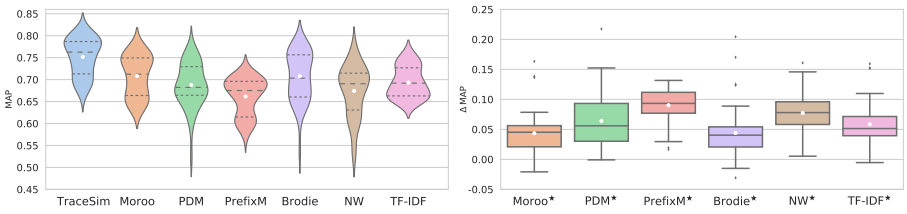
In Figure 9 , we present the results of the same experiments but now with respect to the MAP values obtained by TraceSim and each competing method on the five datasets. In general, TraceSim significantly outperforms the majority of the methods regarding MAP in all datasets. The exceptions are TF-IDF in Gnome and PDM in Eclipse, Netbeans, and JetBrains. Although TraceSim surpasses TF-IDF by 2.35% on average in Gnome, ΔMAP is not statistically significant in this case (the variance varies from -10.46% to 18.87% and the median is relatively close to 0). Regarding PDM, the average of ΔMAP between TraceSim and this method are +0.76% in Eclipse, -0.04% in Netbeans, and +0.25% in JetBrains. However, we consider the performance of these methods comparable since the differences in their results are not significant.

Regarding the RR@1 evaluation metric (Figure 10), TraceSim significantly outperforms most of the methods in all datasets. Similar to the MAP analysis, we do not find statistically significant differences in three exceptional cases: TF-IDF in Gnome, and PDM in both Eclipse and JetBrains. However, we observe distinctive findings in NetBeans: TraceSim significantly underperforms PDM (-1.11% on average), achieving results comparable to PrefixM and Moroo. Even though TraceSim is not dominant in Netbeans regarding RR@1, it consistently achieves competitive RR@1 values across all datasets. For example, on average, TraceSim greatly surpasses PDM by 7.47% in Ubuntu and 12.93% in Gnome.

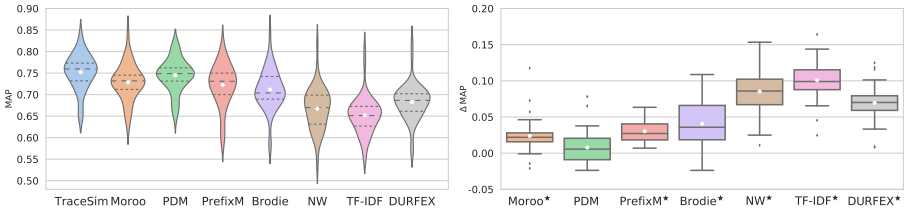
In general, TraceSim consistently achieves competitive performance for each different combination of dataset and metric. Actually, in the majority of the experimented scenarios, it significantly outperforms all competing methods. Our method is surpassed by PDM in a unique scenario (RR@1 on Netbeans). However, in this same dataset, TraceSim substantially outperforms PDM in terms of AUC by 6.94% on average. In fact, none of the competing methods were able to outperform TraceSim across all metrics in a specific dataset. Finally, TraceSim is the only method that consistently performs well on different programming languages. For instance, in terms of MAP, there is no significant difference between PDM and TraceSim on Java datasets (Eclipse, Netbeans, and JetBrains). However, the improvement of TraceSim over PDM regarding MAP is, on average, 6.41% in Ubuntu and 11.99% in Gnome.

5.2 Ablation Study

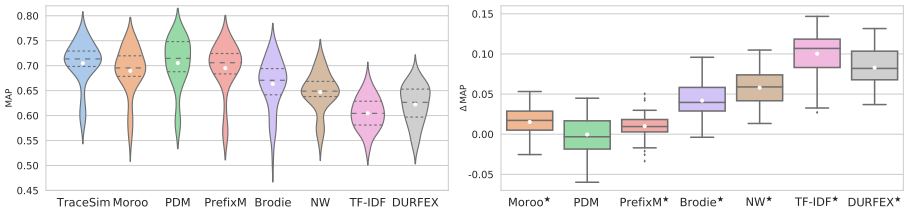
An ablation study aims to assess specific model components by measuring performance degradation when each component is independently removed. In this section, we first conduct an ablation study to assess four important TraceSim components, namely global weight, local weight, the diff$(\cdot)$ function, and normalization. We then evaluate whether the approach to compute mismatch and gap values based on frame weights is more effective than previous strategies proposed in the literature. These two studies are performed only on



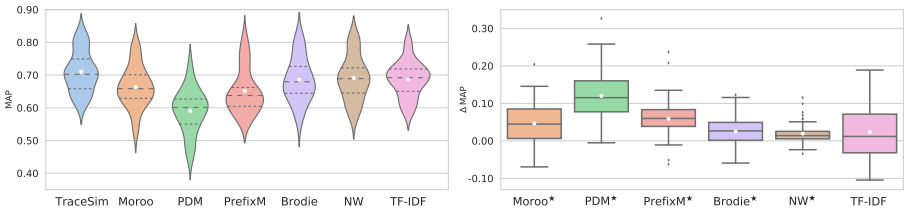
(a) Ubuntu



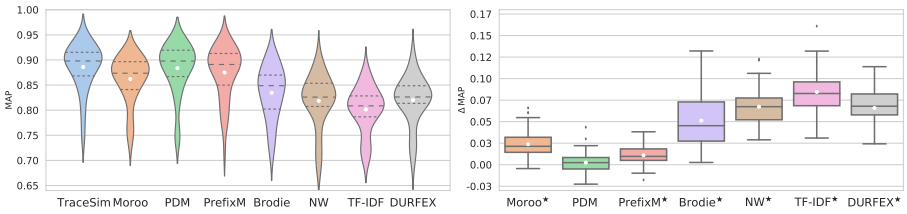
(b) Eclipse



(c) Netbeans



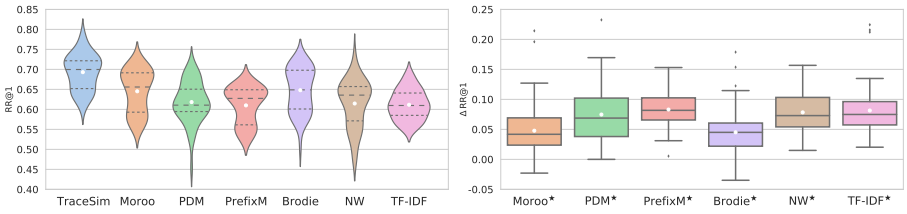
(d) Gnome



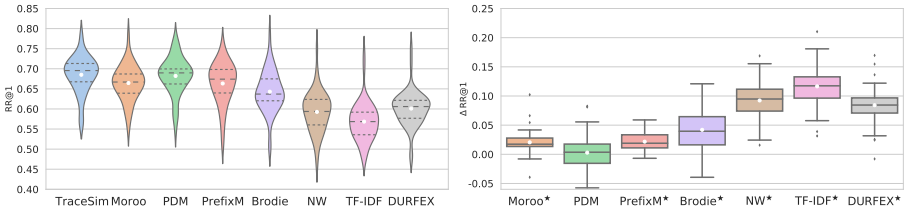
(e) JetBrains

★ $p < 0.01$

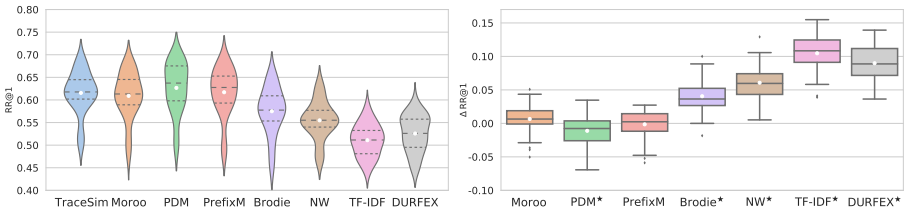
Fig. 9: Results regarding MAP. *Left*: distribution of MAP achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔMAP between TraceSim and each competing method.



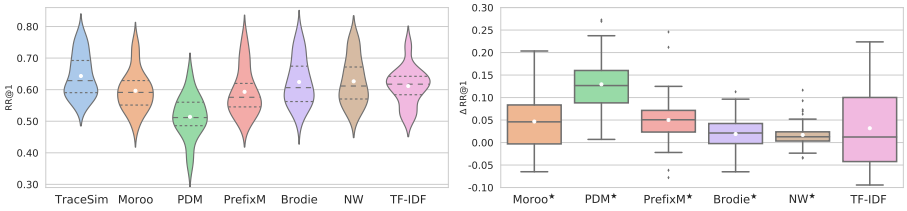
(a) Ubuntu



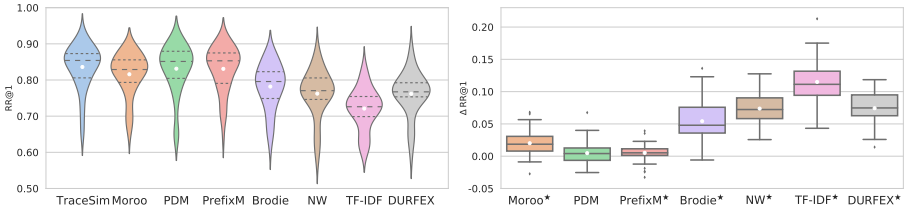
(b) Eclipse



(c) Netbeans



(d) Gnome



(e) JetBrains

★ $p < 0.01$

Fig. 10: Results regarding RR@1. *Left*: distribution of RR@1 achieved by TraceSim and each competing method in all validation sets of each dataset. *Right*: ΔRR@1 between TraceSim and each competing method.

Ubuntu, Eclipse, and Netbeans datasets due to the high computational cost of conducting such extensive experiments on Gnome and JetBrains.

TraceSim Components

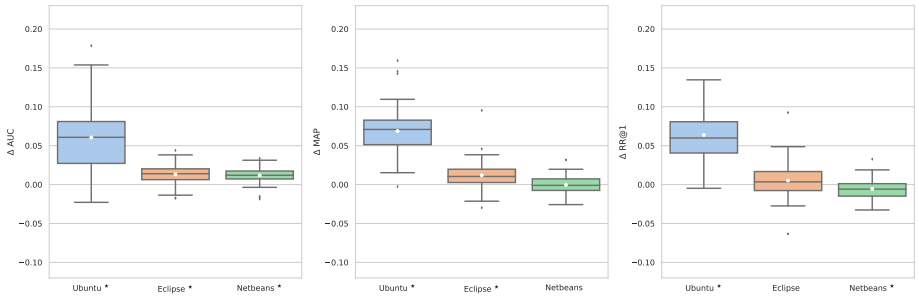
We evaluate four important TraceSim components:

- *Global Weight.* In order to investigate the importance of TF-IDF for TraceSim, we ignore the global weight when computing the weight of a frame. This is achieved by setting $\text{gw}(\cdot) = 1$ in Equation (3).
- *Local Weight.* Although frame positions are known to be valuable features for crash report deduplication, we measure the importance of this information for TraceSim. Thus, the local weight term is ignored by setting $\text{lw}(\cdot) = 1$ in Equation (3).
- The $\text{diff}(\cdot)$ *function.* The difference between the positions of two matched frames is incorporated in PDM and Moroo. However, the corresponding papers do not include a study regarding the importance of this aspect. We ignore the $\text{diff}(\cdot)$ function in the match score function by setting $\text{diff}(\cdot) = 1$ in Equation (8).
- *Normalization.* Many methods (Modani et al, 2007; Lerch and Mezini, 2013; Sabor et al, 2017; Dang et al, 2012) normalize similarity scores, although none of them have investigated the importance of this procedure.

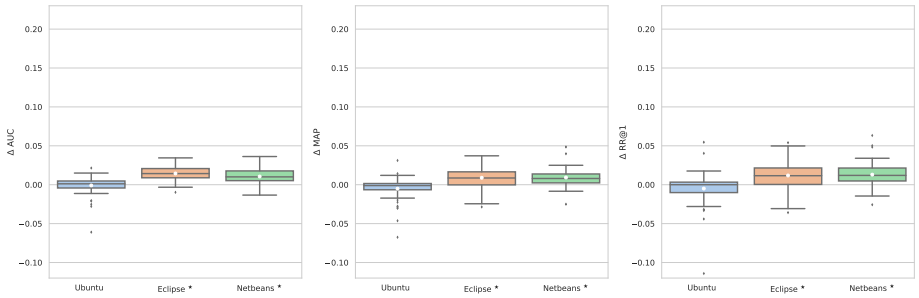
In Figure 11, we depict performance differences (ΔAUC, ΔMAP, and ΔRR@1) between the full TraceSim and its modified versions for which we remove each component listed above.

As shown in Figure 11a, ignoring global weights significantly reduces performance in six out of the nine considered settings. The only three exceptions are RR@1 on Eclipse as well as RR@1 and MAP on NetBeans. Moreover, the only setting for which *Global Weight* deteriorates TraceSim's performance is RR@1 on NetBeans. These results corroborate the hypothesis that the global frequency of subroutines provide valuable information to discriminate important frames. In Figure 11b, we observe that TraceSim performs significantly worse on NetBeans and Eclipse when *Local Weight* is removed. On the Ubuntu dataset, this component appears to have no significant impact. As shown in Figure 11c, the position difference of matched frames significantly improves model performance in all datasets. This corroborates the hypothesis that duplicate stack traces contain important frames in close positions. Finally, in general, performance degrades significantly when normalization is not applied. As illustrated in Figure 11d, this component is not significantly relevant only in three settings: RR@1 on Netbeans, RR@1 on Eclipse, and AUC on Eclipse.

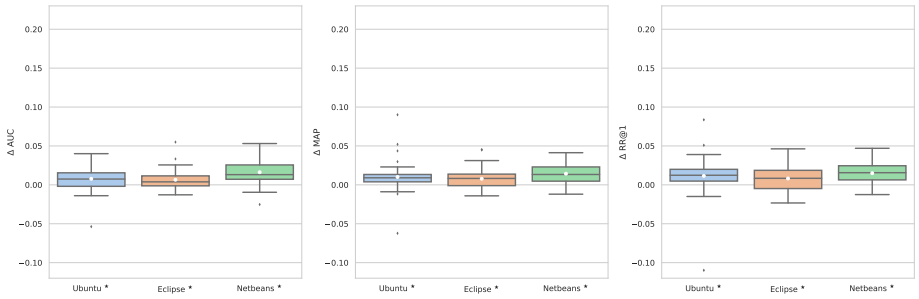
In Appendix A, we additionally report the performance differences between full TraceSim and each of the meaningful combinations that has at least two of four components removed. Overall, the findings are similar to the ones reported in this section.



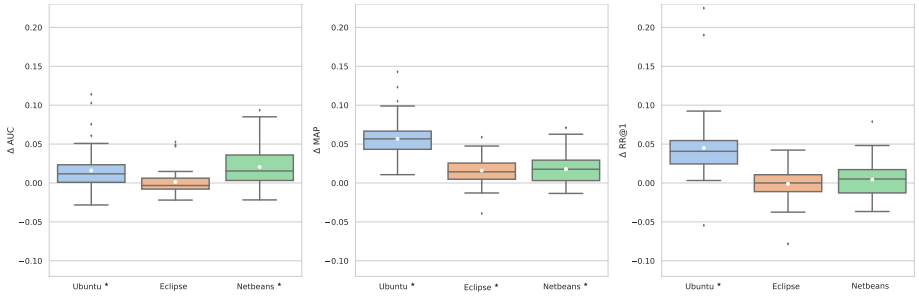
(a) Global Weight removed.



(b) Local Weight removed.



(c) The diff(·) function removed.



(d) Normalization removed.

$^{\star}\ p < 0.01$

Fig. 11: Ablation study results: distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim with different components removed.

Mismatch and Gap Values

In previous works, mismatch and gap values are defined as zero or some other constant. TraceSim, in constrast, defines mismatch and gap values based on frame weights, just as match values. The intuition is that the importance of unmatched frames should also be considered by the alignment algorithm. In order to compare the effectiveness of our strategy, we test all meaningful combinations in which mismatch and gap values are set by one of the following strategies:

- *Zero.* Values are set to zero, so that they have no cost in the optimal alignment.
- *Constant.* Values are constant real numbers tuned by the ML algorithm from the set $\{0.0, 0.1, \ldots, 6.0\}$. These predefined set of values achieved the best and consistent results.
- *Variable.* Gap and mismatch values are given by Equations (6) and (7), respectively. This corresponds to the proposed strategy used in TraceSim for both values: mismatch and gap.

There are nine possible combinations when considering the three aforementioned strategies to set mismatch and gap values individually. However, when Gap=Zero (i.e. gap values are set using the Zero strategy), we have that the mismatch operation is useless regardless of its value, since we can always replace a mismatch by two subsequent gaps with no cost in these cases. Thus, we have that ⟨Mismatch=Constant; Gap=Zero⟩ and ⟨Mismatch=Variable; Gap=Zero⟩ are both equivalent to ⟨Mismatch=Zero; Gap=Zero⟩. This leaves us with seven meaningful strategies. Thus, in the following, we analyse performance differences (ΔAUC, ΔMAP, and ΔRR@1) between the full TraceSim method ⟨Mismatch=Variable; Gap=Variable⟩ and the remaining six strategies.

In Table 5.2, we show whether ΔAUC, ΔMAP, and ΔRR@1 are statistically significant for each one of the six competing strategies in Ubuntu, Eclipse, and Netbeans (the datasets names are abbreviated to U, E, and N, respectively). A ★ symbol in a cell indicates that TraceSim significantly outperforms the strategy on the dataset and metric corresponding to that cell. It is important to highlight that no competing strategy achieves better average performance than TraceSim in these experiments.

First, we focus on the results of the three strategies in which mismatch values are defined as zero ⟨Mismatch=Zero; Gap=*⟩. These results correspond to the first three rows in Table 5.2. The ⟨Mismatch=Zero; Gap=Zero⟩ and ⟨Mismatch=Zero; Gap=Constant ⟩ strategies are equivalent to the strategies used in PDM and Brodie methods, respectively. Overall, the strategy in which mismatch is set to zero negatively affects the method performance. For instance, the ⟨Mismatch=Zero; Gap=Zero⟩ strategy, the best approach among the three ones, significantly degrades the performance in Ubuntu regarding all metrics and in Eclipse and Netbeans in terms of AUC. According to the results, we can conclude that the TraceSim strategy is more effective than the ones employed in Brodie and PDM techniques.

Table 7: Performance differences between TraceSim and the six meaningful strategies to set mismatch and gap values which are statistically significant. Cells marked with a ★ indicate that the performance difference on the corresponding dataset and metric is statistically significant. Due to space constraints, we abbreviate Ubuntu, Eclipse, and NetBeans as U, E, and N, respectively, in the column labels.

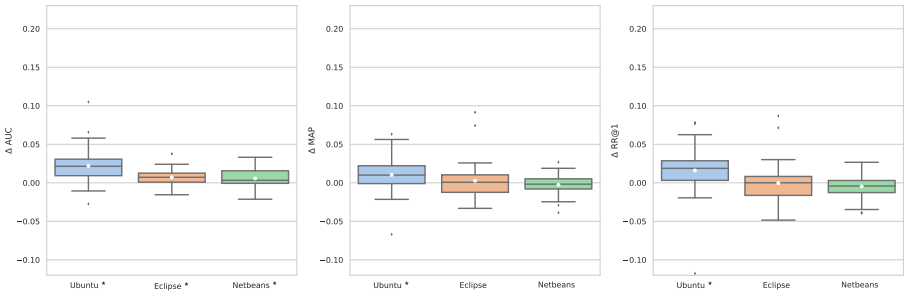
Strategy		**Equiv**	**ΔAUC**			**ΔMAP**			**ΔRR@1**		
Mismatch	**Gap**		**U**	**E**	**N**	**U**	**E**	**N**	**U**	**E**	**N**
Zero	Zero	PDM	★	★	★	★			★		
Zero	Constant	Brodie	★	★	★	★	★		★		
Zero	Variable	–	★	★	★	★	★	★	★		
Constant	Constant	–	★	★	★	★	★		★		
Constant	Variable	–									
Variable	Constant	–	★	★	★	★	★		★	★	

We now consider the results of the three remaining strategies in which mismatch values are set according to either Constant or Variable strategies. Both ⟨Mismatch=Constant; Gap=Constant⟩ (fourth row) and ⟨Mismatch=Variable; Gap=Constant⟩ (last row) strategies perform significantly worse in, respectively, six and seven out of the nine evaluation settings. On the other hand, the ⟨Mismatch=Constant; Gap=Variable⟩ strategy presents negligible effect on model performance, resulting in no significant difference in any evaluation setting. After analysing these results more carefully, we found that the constant mismatch value was set to high values in most of the cases by the TPE technique. More specifically, this value was equal to or greater than 2.0 in 126 out of 150 chunks, i.e. 84% of the cases. Since the upper bound of a gap value is 1.0 (see Equation (6)), a mismatch value equal to or greater than 2.0 means that the alignment algorithm will basically avoid mismatches. Recall that in TraceSim two subsequent gaps can always replace a mismatch with no effect in the optimal alignment cost. Thus, the ⟨Mismatch=Constant; Gap=Variable⟩ strategy, when using such high mismatch values, is equivalent to TraceSim. Overall, the results corroborate the hypothesis that gaps should be prioritized over mismatches, since gaps are more flexible. Moreover, we conclude that the proposed strategy to set mismatch and gap values based on frame importance is relevant to TraceSim's performance.

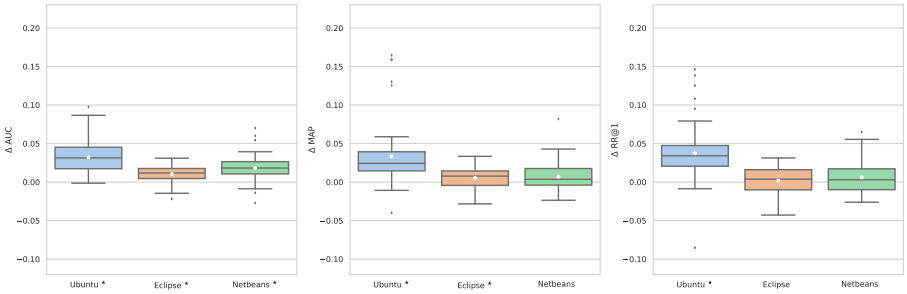
In addition to Table 5.2, for more details, we provide box-and-whiskers plots of the performance differences between TraceSim and the six strategies in Figure 12 and Figure 13.

5.3 Time Efficiency

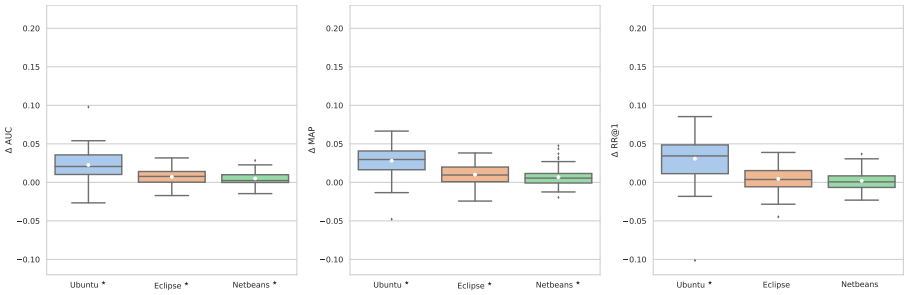
The comparison of a query to candidates within a dataset is the most critical efficiency issue in crash report deduplication. Due to the use of an inverted index, information retrieval methods are more efficient than matching algorithms to



(a) Mismatch=Zero/Gap=Zero



(b) Mismatch=Zero/Gap=Constant

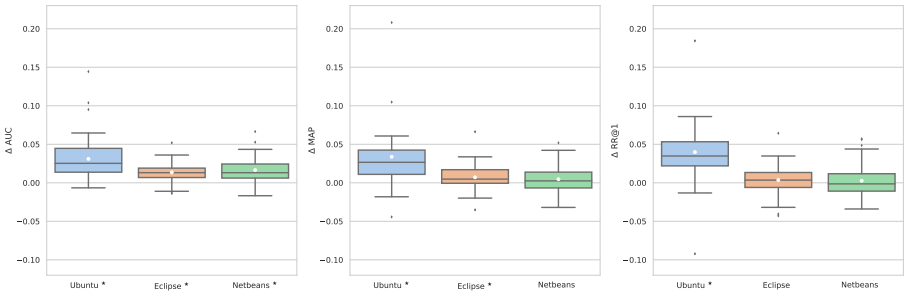


(c) Mismatch=Zero/Gap=Variable

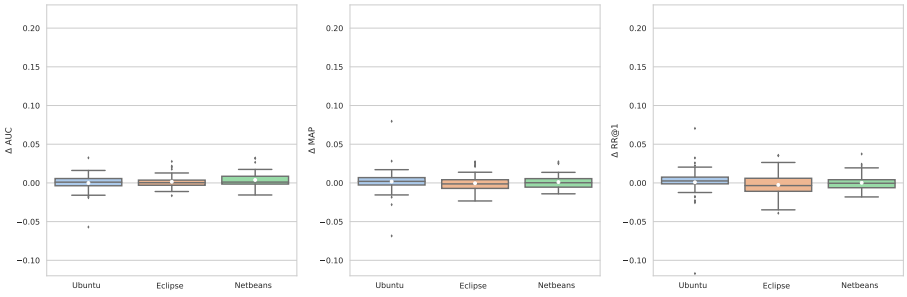
⋆ $p < 0.01$

Fig. 12: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between complete TraceSim and three strategies in which mismatch values are defined as zero.

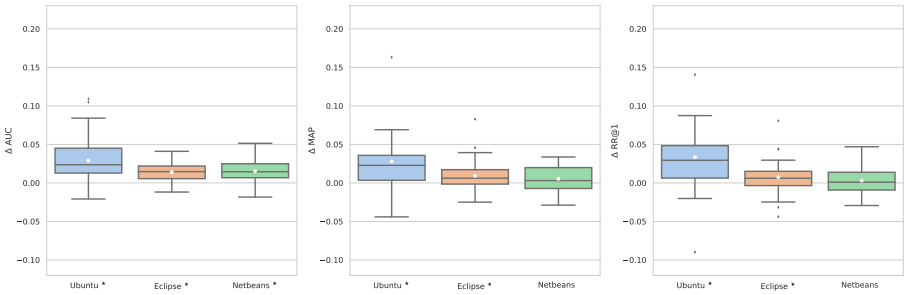
generate the ranked list of a query q. In one hand, the complexity of information retrieval methods with inverted index is $O(|qb| \log |V| d)$, where $|qb|$ is the set of subroutines in the query, $|V|$ is the vocabulary size, d is equal to the number of candidates in $C(q)$. In the other hand, the complexity of matching algorithms is $O(|q||c|_{max}|C(q)|)$, where $|q|$ is the query length, and $|c|_{max}$ is the longest candidate in $C(q)$. In real applications, d tends to be much smaller than $|C(q)|$



(a) Mismatch=Constant/Gap=Constant



(b) Mismatch=Constant/Gap=Variable



(c) Mismatch=Variable/Gap=Constant

⋆ $p < 0.01$

Fig. 13: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between complete TraceSim and three strategies in which mismatch values are set according to either Constant or Variable strategies.

since information retrieval techniques only compute the similarity of candidates that contain subroutines shared with the query. This explains the considerable superiority of information retrieval techniques over matching algorithms in terms of efficiency.

In practice, two simple approaches can be employed to speed up matching algorithms. The first one consists of using time windows to reduce the number

Table 8: Throughput (queries / second) of TraceSim with and without time window, TF-IDF, and Moroo in Gnome and JetBrains.

Method	Gnome	JetBrains
TraceSim without Time window	0.5319	0.1080
TraceSim with Time window	2.6897	0.2569
TF-IDF	11.7687	1.9476
Mooro	8.8822	0.8051

of candidates to be considered for deduplication. The second approach, called *re-ranking*, first creates a temporary ranked list based on information retrieval techniques. Then it recalculates the similarity score of the top-k candidates in the list using a more complex algorithm.

We examine the throughput of TraceSim (with and without time window), TF-IDF, and Moroo (a re-ranking technique) on the two largest datasets: Gnome and JetBrains. These techniques are executed on one of the original validation sets in a controlled and homogeneous environment[16]. Excluding the reports submitted after the validation sets, the experiment setups in Gnome and JetBrains contain 216,646 and 901,015 reports, respectively. As we can observe, the number of reports is very close to the total number of reports in the datasets (see Table 1).

In Table 8, we show the throughput (queries / second) of TraceSim with and without time window, TF-IDF, and Moroo in Gnome and JetBrains data sets. The time window approach improves the algorithm throughput by around 5.0 and 2.5 times in Gnome and JetBrains, respectively. However, even using a time frame, TF-IDF is substantially more efficient than TraceSim – the speedup is approximately 4.37 and 7.58 in Gnome and JetBrains. As expected, Moroo is slower than TF-IDF but the gap is much smaller in comparison to TraceSim – TF-IDF is 1.32 and 2.41 times faster than Moroo in Gnome and JetBrains.

In brief, it is difficult to determine whether the throughput of TraceSim presented in Table 8 is viable or not for a particular software project since a throughput of 2.62 or 0.25 may be satisfactory depending on the application. In cases that TraceSim is not viable due to its run time, besides calibrating the time window, re-ranking could be employed to accelerate it. However, more investigation is needed to figure out whether re-ranking would degrade the quality of deduplication.

6 Threats to Validity

In this section, we discuss some threats to the validity of our study.

Quality of labeled data. Duplicate crash reports are identified by human triagers. Since this is not a trivial task, reports might be incorrectly classified

[16] The experiments in Section 5.1 and 5.2 were run in a shared and heterogeneous environment. Therefore, it is difficult to compare the run times based on these experiments.

as non-duplicate or inserted into an incorrect bucket. To mitigate this threat, we avoided using the most recent reports from the repositories. The assumption is that most incorrect labels tend to be corrected over time. Moreover, we employed data from well-known applications that have been used in literature for crash deduplication and duplicate bug report detection.

Subject selection bias. The performance of the considered methods is significantly dependent on data. Thus, the superiority of a method over other techniques might differ concerning other software projects. To mitigate this problem, we have conducted our experiments on five distinct software projects, which contain stack traces from different programming languages (C/C++ and Java). Moreover, four of our datasets come from different open-source applications while the other is an industrial dataset from IntelliJ Platform products.

Stack trace extraction. In the four open source datasets, we extracted the stack traces from the textual data of bug reports using different parsers for each programming language. Since textual data is unstructured, parsers might: extract only partial stack trace information, or miss stack traces (false negatives), or wrongly detect a fraction of texts as correct stack traces (false positives). We mitigate these possible issues by using parsers that are well-known in the literature by the community.

Competing method implementations. Except for the work of Campbell et al (2016), existing studies did not make available their implementations and/or the data used for experimentation. Hence, we had to implement all the baselines and state-of-the-art methods. Even though we have carefully followed the technique descriptions in the papers, our implementations might not fully match the originals since crucial components and preprocessing steps of the techniques might not have been described with complete accuracy, or even been reported in the study at all.

7 Conclusions

In this paper, we proposed TraceSim, a novel technique for crash report deduplication. TraceSim computes the similarity between a pair of stack traces by finding the optimal global alignment of their frame sequences. To compute the alignment score, we assign weights to frames that indicate their discriminativeness for stack trace comparison. These weights depend on two factors: 1) the position of the frames, and 2) the frequency of the subroutines in the dataset. The influence of these factors on the similarity is regulated by parameters that are learned using ML algorithms. Unlike previous techniques, the alignment scores are influenced by the weights of all frames, matched and unmatched, in the stack traces.

TraceSim and seven competing methods were experimentally evaluated on five datasets (four generated from open-source projects and one derived from industrial data) using a new methodology that combines ranking and binary classification metrics. Except for industrial project data, the full evaluation

framework – including datasets and the source code of methods and evaluation methodology – is freely available online. We have found that TraceSim outperforms the majority of the existing methods in the literature. Moreover, our method performs consistently well in all distinct scenarios including datasets with distinct programming languages. In summary, compared to the previously proposed methods:

1. TraceSim distinguishes duplicate reports from non-duplicate ones more accurately.
2. TraceSim assigns reports to their correct buckets more often.
3. TraceSim generates better recommendation lists when the system needs human assistance.

Furthermore, we conducted an ablation study to investigate the effectiveness of TraceSim components and its scheme to compute mismatch and gap values. The results corroborated that the frame position (local weight) and document frequency (global weight) are valuable for crash report deduplication, as well as normalization and the use of the position difference of the matched frames. Finally, the experimental results confirmed the hypothesis that the rarity and the position of the frames should be considered for computing mismatch and gap values.

In terms of run time, TraceSim is similar to previously proposed sequence matching algorithms. However, due to the use of inverted index, information retrieval methods are more efficient in comparing a query to all n reports within a repository. Besides employing re-ranking, we plan to investigate additional approaches to reduce the computational cost without negatively affecting TraceSim's performance.

References

Ahmed I, Mohan N, Jensen C (2014) The impact of automatic crash reports on bug triaging and development in mozilla. In: Proceedings of The International Symposium on Open Collaboration, Association for Computing Machinery, New York, NY, USA, OpenSym '14, p 1–8, DOI 10.1145/2641580.2641585, URL https://doi.org/10.1145/2641580.2641585

Banerjee S, Syed Z, Helmick J, Culp M, Ryan K, Cukic B (2017) Automated triaging of very large bug repositories. Information and Software Technology 89:1 – 13, DOI https://doi.org/10.1016/j.infsof.2016.09.006, URL http://www.sciencedirect.com/science/article/pii/S0950584916301653

Bartz K, Stokes JW, Platt JC, Kivett R, Grant D, Calinoiu S, Loihle G (2008) Finding similar failures using callstack similarity. In: Proceedings of the Third Conference on Tackling Computer Systems Problems with Machine Learning Techniques, USENIX Association, Berkeley, CA, USA, SysML'08, pp 1–1, URL http://dl.acm.org/citation.cfm?id=1855895.1855896

Bergstra J, Yamins D, Cox DD (2013a) Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In: Proceedings of the 12th Python in Science Conference, Citeseer, pp 13–20

Bergstra J, Yamins D, Cox DD (2013b) Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, JMLR.org, ICML'13, p I–115–I–123

Brodie M, Ma S, Lohman G, Mignet L, Modani N, Wilding M, Champlin J, Sohn P (2005) Quickly finding known software problems via automated symptom matching. In: Second International Conference on Autonomic Computing (ICAC'05), pp 101–110, DOI 10.1109/ICAC.2005.49

Campbell JC, Santos EA, Hindle A (2016) The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '16, pp 269–280, DOI 10.1145/2901739.2901766, URL http://doi.acm.org/10.1145/2901739.2901766

Chierichetti F, Kumar R, Pandey S, Vassilvitskii S (2010) Finding the jaccard median. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, SIAM, pp 293–311

Dang Y, Wu R, Zhang H, Zhang D, Nobel P (2012) Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '12, pp 1084–1093, URL http://dl.acm.org/citation.cfm?id=2337223.2337364

Deza MM, Deza E (2016) Encyclopedia of Distances, 4th edn. Springer Berlin Heidelberg, DOI 10.1007/978-3-642-00234-2_1

Dhaliwal T, Khomh F, Zou Y (2011) 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, IEEE Computer Society, Washington, DC, USA, ICSM '11, pp 333–342, DOI 10.1109/ICSM.2011.6080800, URL https://doi.org/10.1109/ICSM.2011.6080800

Ebrahimi N, Trabelsi A, Islam MS, Hamou-Lhadj A, Khanmohammadi K (2019) An hmm-based approach for automatic detection and classification of duplicate bug reports. Information and Software Technology 113:98–109

Gehan EA (1965) A generalized wilcoxon test for comparing arbitrarily singly-censored samples. Biometrika 52(1/2):203–223, URL http://www.jstor.org/stable/2333825

Glerum K, Kinshumann K, Greenberg S, Aul G, Orgovan V, Nichols G, Grant D, Loihle G, Hunt G (2009) Debugging in the (very) large: Ten years of implementation and experience. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA, SOSP '09, p 103–116, DOI 10.1145/1629575.1629586, URL https://doi.org/10.1145/1629575.1629586

Kampstra P (2008) Beanplot: A boxplot alternative for visual comparison of distributions. Journal of Statistical Software, Code Snippets 28(1):1–9, DOI 10.18637/jss.v028.c01, URL https://www.jstatsoft.org/v028/c01

Kim S, Zimmermann T, Nagappan N (2011) Crash graphs: An aggregated view of multiple crashes to improve crash triage. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), IEEE,

pp 486–493

Koopaei NE, Hamou-Lhadj A (2015) Crashautomata: An approach for the detection of duplicate crash reports based on generalizable automata. In: Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, IBM Corp., USA, CASCON '15, p 201–210

Lerch J, Mezini M (2013) Finding duplicates of your yet unwritten bug report. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '13, pp 69–78, DOI 10.1109/CSMR.2013.17, URL http://dx.doi.org/10.1109/CSMR.2013.17

Manning CD, Schütze H (1999) Foundations of Statistical Natural Language Processing. The MIT Press, Cambridge, Massachusetts, URL http://nlp.stanford.edu/fsnlp/

Miller FP, Vandome AF, McBrewster J (2009) Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance. Alpha Press

Modani N, Gupta R, Lohman G, Syeda-Mahmood T, Mignet L (2007) Automatically identifying known software problems. In: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop, IEEE Computer Society, Washington, DC, USA, ICDEW '07, pp 433–441, DOI 10.1109/ICDEW.2007.4401026, URL http://dx.doi.org/10.1109/ICDEW.2007.4401026

Moroo A, Aizawa A, Hamamoto T (2017) Reranking-based crash report deduplication. In: He X (ed) SEKE '17, pp 507–510, DOI 10.18293/SEKE2017-135

Needleman S, Wunsch C (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48(3):443–453, DOI 10.1016/0022-2836(70)90057-4

Putatunda S, Rama K (2018) A comparative analysis of hyperopt as against other approaches for hyper-parameter optimization of xgboost. In: Proceedings of the 2018 International Conference on Signal Processing and Machine Learning, Association for Computing Machinery, New York, NY, USA, SPML '18, p 6–10, DOI 10.1145/3297067.3297080, URL https://doi.org/10.1145/3297067.3297080

Rakha MS, Bezemer C, Hassan AE (2018) Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. IEEE Transactions on Software Engineering 44(12):1245–1268, DOI 10.1109/TSE.2017.2755005

Sabor KK, Hamou-Lhadj A, Larsson A (2017) DURFEX: A feature extraction technique for efficient detection of duplicate bug reports. In: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017, IEEE, pp 240–250, DOI 10.1109/QRS.2017.35

Schroter A, Schröter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), IEEE, pp 118–121

Sellers PH (1974) On the theory and computation of evolutionary distances. SIAM Journal on Applied Mathematics 26(4):787–793

Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, ASE '11, pp 253–262, DOI 10.1109/ASE.2011.6100061, URL http://dx.doi.org/10.1109/ASE.2011.6100061

Waskom M (2020) mwaskom/seaborn. DOI 10.5281/zenodo.592845, URL https://doi.org/10.5281/zenodo.592845

A Additional ablation study results

In this appendix, we expand the ablation study in which Global Weight, Local Weight, the diff(·) Function, and normalization are removed. We depict ΔAUC, ΔMAP, and ΔRR@1 between the original TraceSim and each possible configuration that has not more than two components enabled in Figures 14–21.

The following configurations are not reported:

1. *TraceSim without Global Weight and Local Weight.* In this case, frame weights are always equal to 1. Since the normalization was designed based on variable frame weights, the normalization loses its effectiveness.
2. *TraceSim without Global Weight, Local Weight, and the* diff(·) *Function.* Similarly to the previous configuration, the normalization is not effective because the frame weights are constants.
3. *TraceSim without Global Weight, Local Weight, normalization and the* diff(·) *Function.* This configuration is equivalent to NW algorithm in which the match, mismatch and gap values are set to 1.0, 2.0, and 1.0, respectively.

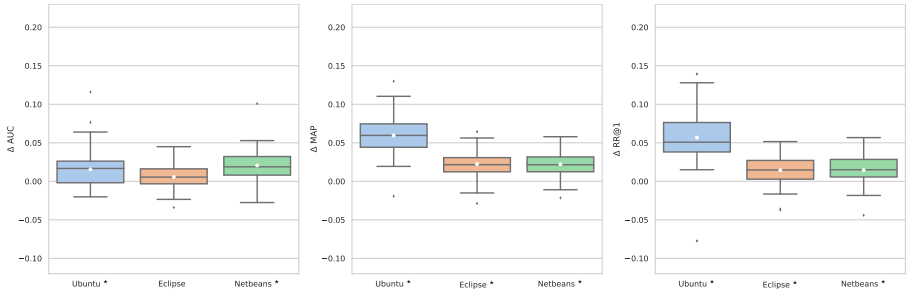


Fig. 14: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without the* diff(·) *Function and Normalization.*

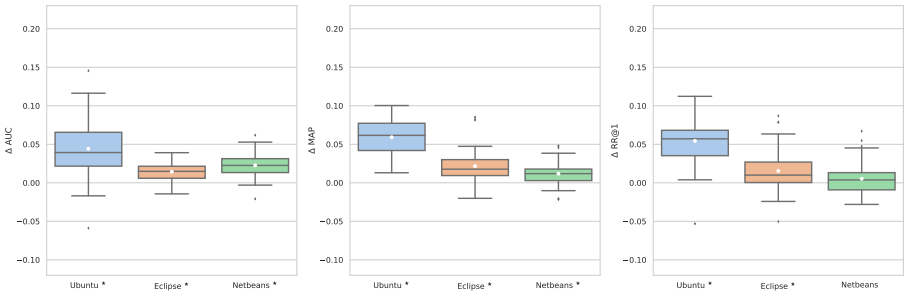


Fig. 15: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight and the* diff$(\cdot)$ Function.

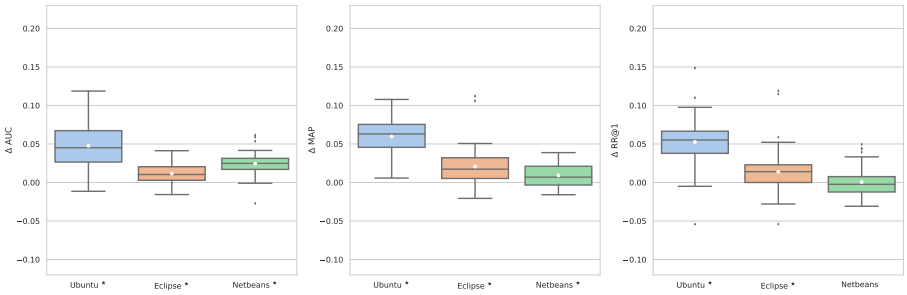


Fig. 16: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight and Normalization.*

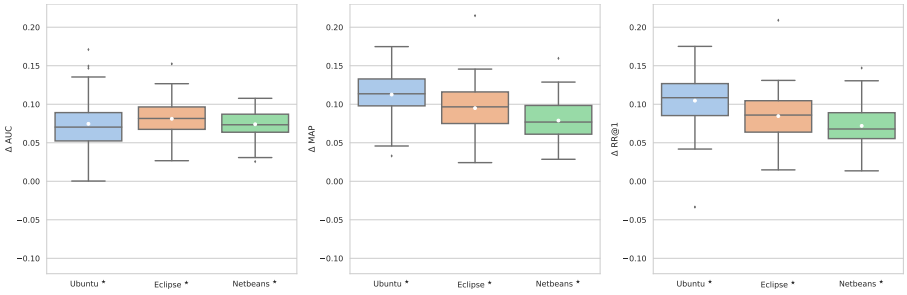


Fig. 17: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight, Local Weight and Normalization.*

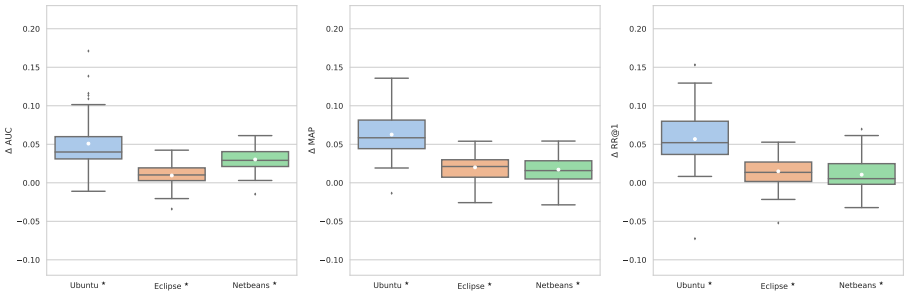


Fig. 18: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Global Weight, the* diff$(\cdot)$ *Function, and Normalization.*

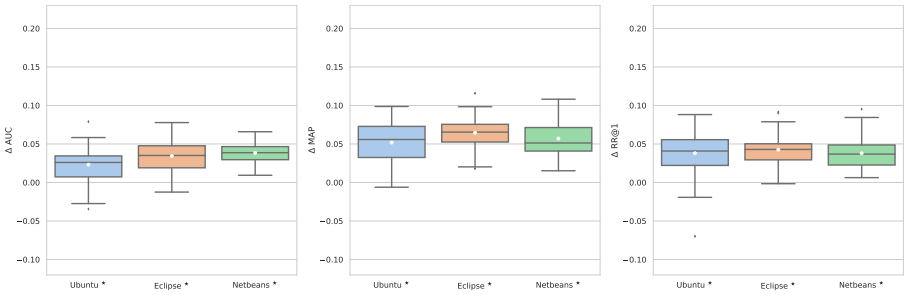


Fig. 19: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight and Normalization.*

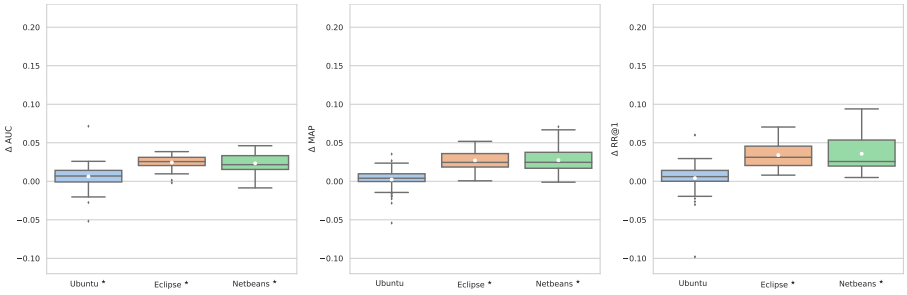


Fig. 20: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight and the* diff$(\cdot)$ Function.

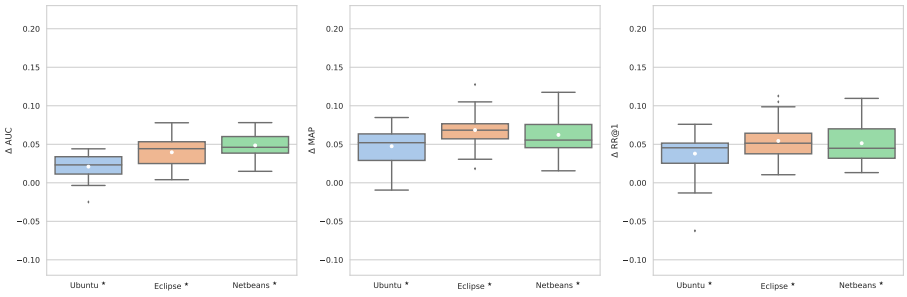


Fig. 21: Distributions of ΔAUC (left), ΔMAP (middle) and ΔRR@1 (right) between full TraceSim and TraceSim *without Local Weight, the* diff$(\cdot)$ *Function and Normalization.*