DeepLSH: Deep Locality-Sensitive Hash Learning for Fast and Efficient Near-Duplicate Crash Report Detection

Youcef Remil
INSA Lyon
Infologic R&D
26500 Bourg-Lès-Valence, France
yre@infologic.fr

Anes Bendimerad
Infologic R&D
26500 Bourg-Lès-Valence, France
abe@infologic.fr

Romain Mathonat
Infologic R&D
26500 Bourg-Lès-Valence, France
rma@infologic.fr

Chedy Raissi
Riot Games
018937 Marina One West Tower
Singapore
chedy.raissi@inria.fr

Mehdi Kaytoue
INSA Lyon
Infologic R&D
26500 Bourg-Lès-Valence, France
mka@infologic.fr

ABSTRACT

Automatic crash bucketing is a crucial phase in the software development process for efficiently triaging bug reports. It generally consists in grouping similar reports through clustering techniques. However, with real-time streaming bug collection, systems are needed to quickly answer the question: *What are the most similar bugs to a new one?*, that is, efficiently find near-duplicates. It is thus natural to consider nearest neighbors search to tackle this problem and especially the well-known locality-sensitive hashing (LSH) to deal with large datasets due to its sublinear performance and theoretical guarantees on the similarity search accuracy. Surprisingly, LSH has not been considered in the crash bucketing literature. It is indeed not trivial to derive hash functions that satisfy the so-called *locality-sensitive property* for the most advanced crash bucketing metrics. Consequently, we study in this paper how to leverage LSH for this task. To be able to consider the most relevant metrics used in the literature, we introduce DeepLSH, a Siamese DNN architecture with an original loss function, that perfectly approximates the locality-sensitivity property even for Jaccard and Cosine metrics for which exact LSH solutions exist. We support this claim with a series of experiments on an original dataset, which we make available.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **Computing methodologies → Randomized search**; • **Theory of computation → Theory of randomized search heuristics**.

KEYWORDS

Crash deduplication, Stack trace similarity, Approximate nearest neighbors, Locality-sensitive hashing, Siamese neural networks

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639146

ACM Reference Format:
Youcef Remil, Anes Bendimerad, Romain Mathonat, Chedy Raissi, and Mehdi Kaytoue. 2024. DeepLSH: Deep Locality-Sensitive Hash Learning for Fast and Efficient Near-Duplicate Crash Report Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639146

1 INTRODUCTION

Collection and triage of runtime errors following a software release are integral parts of a standard quality process (e.g., Windows Error Reporting [12], Mozilla Crash Reporter [31]). At our company, which specializes in editing and installing b2b Enterprise Resource Planning (ERP) systems, we receive a daily influx of over ten thousand of automatic and user-generated reports. Each report comprises a Java stack trace and relevant contextual information. It is important to note that not all crash reports hold the same level of priority: Rare occurrences indicate unexpected shutdowns, more frequent ones involve GUI issues that obstruct end-users from completing their tasks, while the majority fall into the "silent" category. The latter typically represents bugs that either have workarounds found by users (thus limiting their impact) or background process failures that may take days to notice but have significant consequences. Hence, it is imperative to promptly address such bugs.

While extreme problems are rare and easy to identify and prioritize, it remains challenging to sort, rank and assign other reports to developers (or simply ignore them). Indeed, many different reports may actually imply a single root cause and a bug can produce slightly different stack traces known as *near-duplicates* [8]. Therefore, it is highly valuable to group similar crashes into buckets to accelerate the crash investigation process [10]. Reporting systems use a range of heuristics and manually developed rules to organize crash reports into categories, ideally each referring to the same bug [12]. However, in many cases, it may assign crash reports caused by the same bug to multiple buckets [12]. Thus, various alternatives address the *stack trace-based report deduplication problem* by designing custom and accurate similarity measures between stack traces, relying mostly on string and graph matching (e.g., edit distance, prefix match and LCSS) [5, 10, 20] and information retrieval (e.g., TF-IDF and N-grams) [22, 33]. Other metrics take into

account specific characteristics of stack traces, such as the distance to the top frame or alignment between matched frames) [8, 30, 38]. In the aforementioned studies, similarity measures are generally embedded in clustering algorithms, but this comes with several drawbacks: First, it needs numerous similarity calculations when assigning a new stack trace to a cluster. Second, clusters are not stable over time and should be recalculated frequently without losing links to actual bug tickets that have been previously created. It can also be difficult to set the various parameters that need to be tuned.

In our company, we aim to process reports in quasi-real time and enhance our maintenance processes. An essential aspect of this objective is efficiently identifying the nearest neighbors for each crash report. We need to quickly determine if a corresponding ticket has already been created, allowing us to update its statistics, or if a new ticket needs to be generated. This becomes even more crucial when end-users directly contact us via email, phone, or assistance tickets. We must perform the same checks to ascertain whether similar issues have been encountered before and if any temporary or permanent solutions have been provided. However, the computational demands of searching for nearest neighbors are generally prohibitive. For instance, we faced a significant challenge when conducting linear scans, which took approximately 10 hours to compare 1,000 stack traces against a pool of 100,000 stack traces using the similarity metric proposed by Brodie et al.[5]. This well-known observation within the field of data mining has led us to explore the concept of *approximate nearest neighbors search* (ANN)[28].

Hashing is a popular technique for ANN [28], and particularly LSH, allowing to search for approximate nearest neighbors in constant time. LSH satisfies the *locality-sensitive property*, that is, similar items are expected to have a higher probability to be mapped into the same hash code (or hash bucket) than dissimilar items [39]. Most importantly, LSH provides guarantees on the search accuracy that is, *the probability for two stack traces having randomly the same hash function is equal to their similarity* and the collision probability of being hashed into at least one bucket can be simply and fully controlled with two key parameters representing the user's desired search precision and recall. The more accurate LSH is, the larger its hash tables are. Fortunately, LSH is known for its computational and storage efficiency, as well as its sublinear search performance [16].

LSH remains surprisingly unexplored in the crash bucketing literature, despite its potential benefits. It is indeed not trivial to derive hash functions that guarantee the *locality-sensitive property* for the most advanced metrics of crash bucketing. Generating hash functions that meet this property is a non-trivial task. Although several LSH function families have been proposed, each for estimating one and only one conventional similarity/distance measure, e.g., Min-Hash function for Jaccard coefficient [4] and Sign-Random-Projection (Sim-Hash) function for angular distance [6], etc., a generalized procedure for applying LSH to various similarity measures remains ambiguous and theoretically complex. More specifically, there is currently no systematic procedure for deriving a family of LSH functions for any given similarity measure.

Exploring the application of LSH to custom similarities for crash deduplication is a novel area that we delve into. We propose to learn these hash functions in a supervised manner to mimic any given similarity measure while incorporating the locality-sensitive hashing component into the model learning process. We draw inspiration from the field of *learn to hash* techniques [11, 21, 23, 25, 26, 40], which effectively reduce the dimensionality of input data representations while preserving similarity. In fact, we aim to leverage the strengths of both LSH and Learn to Hash approaches. Our proposed model generates a hash code, with LSH guarantees, that is shared by the nearest neighbors of the input stack trace. To the best of our knowledge, this is the first *similarity-agnostic* method that utilizes hashing for the crash deduplication problem. It is important to note that our objective is not to introduce a new similarity measure, but rather to enable existing measures to scale effectively.

Contribution. Our contribution is three-fold. (i) Aiming to overcome the problem of deriving LSH functions for stack-trace similarity measures, we propose a generic approach dubbed DeepLSH that learns and provides a family of binary hash functions that perfectly approximate the *locality-sensitive property* to retrieve efficiently and rapidly near-duplicate stack traces. (ii) Technically, we design a deep Siamese neural network architecture to perform end-to-end hashing with an original objective loss function based on the locality-sensitive property preserving with appropriate regularizations to cope with the binarization problem of optimizing non-smooth loss functions. (iii) We demonstrate through our experimental study the effectiveness and scalability of DeepLSH to yield near-duplicate crash reports under a dozen of similarity metrics. We successfully compare to standard LSH techniques (MinHash and SimHash), and the most relevant deep hashing baseline [14] on a large real-world dataset that we make available.

Main findings. (i) DeepLSH demonstrates exceptional convergence to the locality sensitive property for all studied stack-trace similarity measures: it effectively maintains the LSH guarantees, exhibiting high precision/recall. (ii) DeepLSH consistently achieves satisfactory search accuracy, with a recall rate and a ranking quality up to 0.9 for unseen stack traces. (iii) DeepLSH almost matches MinHash (for Jaccard) and outperforms SimHash (for Cosine) in search performance while still generalizing to other complex similarity metrics. (iv) DeepLSH is highly scalable and can retrieve near-duplicate crash reports in a constant time (an average of 24 seconds on 100 millions queries) compared to an exact K-NN approach which takes hours to perform a linear scan. (v) Our end-to-end hashing approach combining LSH and learn-to-hash has shown to be significantly more accurate than using only learn-to-hash or LSH performed separately on learn to hash results, as demonstrated by comparison to the most identified relevant baseline [14].

2 RELATED WORK

Our work encompasses two research areas which will be discussed in this section: (1) the approximate nearest neighbor search through hashing techniques, and (2) custom similarity measures for the stack trace deduplication problem.

Hashing for ANN search. Locality-sensitive hashing has been widely studied by the theoretical computer science community. Its main aspect focuses on the generation of a family of random hash functions that meet the locality-sensitive property for conventional similarity measures [4, 6, 9, 16, 34]. Particularly, Min-Hash (or min-wise independent permutations) [4] is an LSH function designed specifically for Jaccard similarity. Sim-Hash [6, 34] is another popular technique whose aim is to estimate angular similarities such as

Cosine. Sim-Hash has been adopted by Google [34] and it is often used in text processing applications to compare between documents. Both techniques cannot, however, be applied to estimate other similarity metrics besides Jaccard or Cosine. LSH has garnered limited interest within the software engineering community, mainly being solicited for tasks like code search [37] and clone detection [17] through the utilization of well-known LSH functions such as Min-Hash and Hamming LSH. However, it remains an unexplored area in the domain of crash-deduplication, primarily due to the fact that the currently used metrics do not facilitate the application of conventional and well-established LSH functions.

Designing LSH functions for any given similarity metric remains ambiguous and theoretically challenging, as there is no established method for deriving a set of LSH hash functions for a specific similarity measure. On the other hand, the concept of learn to hash has become the focus of many learning-based hashing methods especially for the computer vision community [13, 14, 21, 24–26, 40, 41]. These methods are primarily designed for searching image similarity and have proven to be highly effective in reducing the dimensionality of input data representations while preserving their similarities. However, they do not meet our main objective, which consists in an end-to-end procedure to retrieve near-duplicate data objects with guarantees. They do not reveal a systematic way to construct hash tables from the resulting hash codes, and neither do they control the trade-off between recall and precision using key parameters as does LSH. Alternatively, we take benefit from both worlds, i.e., LSH and Learn to Hash, by proposing an end-to-end procedure that incorporates the LSH component in the learning process. We have identified a similar baseline approach in [14] that proposes a different methodology compared to ours, consisting of a deep hash coding neural network combined with Hamming LSH fitted on the resulting hash vectors to retrieve near-duplicate images in a large database. The authors first proposed a constrained loss function without incorporating the locality-sensitive property, and then performs discretization on continuous hash vectors to carry out the Hamming LSH separately from the model. We show through our experiments in Sec. 5 by adapting this two-step approach on stack traces, that it leads to a considerable search performance degradation compared to our approach DeepLSH.

Stack trace similarities. We report research studies that tackle the crash report deduplication problem using stack trace similarity functions. Lerch and Mezini [22] employed the TF-IDF-based scoring function from Lucene library [27]. Sabor et al. [33] proposed DURFEX system which uses the package name of the subroutines and then segment the resulting stack traces into N-grams to compare them using the Cosine similarity. Some alternative techniques propose to compute the similarity using derivatives of the Needleman-Wunsch algorithm [32]. In [5], Brodie et al. suggested to adjust the similarity based on the frequency and the position of the matched subroutines. Dang et al. [8] proposed a new similarity measure called PDM in their framework Rebucket to compute the similarity based on the offset distance between the matched frames and the distance to the top frame. More recently, TraceSim [38] has been proposed to take into consideration both the frame position and its global inverse frequency. Moroo et al [30] present an approach that combines TF-IDF coefficient with PDM. Finally we outline some earlier approaches that used edit distance as it is equivalent to optimal global alignment [1, 29]. Note that our approach DeepLSH does not propose a new similarity measure and does not question the effectiveness or compete against these existing measures, but it is complementary to them. We demonstrate that DeepLSH model is able to estimate all these measures with the purpose of providing a scalable way to yield approximate near-duplicate stack traces w.r.t these custom similarity functions.

3 BACKGROUND AND PROBLEM DEFINITION

3.1 Crash reports and stack-trace dataset

Software often contains bugs that can lead to crashes and errors. In the following, we use both terms interchangeably to refer to instances of application crashes, where the system becomes unresponsive, as well as errors arising from background tasks or error pop-ups presented to end-users. Each of these issues is accompanied by a Java stack trace and a run-time context (software/OS/database version, timestamp, etc.) [36]. A stack trace is a detailed report of the executed methods and their associated packages during a crash. Stack traces can be retrieved through system calls in many programming languages. In Java, the stack trace lists methods in descending order, with the top of the stack trace representing the most inner call. This is illustrated with a crash report from a software product in Fig. 1. We define the stack-trace dataset as set of N stack traces $\mathcal{D} = \{s_1, s_2, ..., s_N\}$.

```
id: 16377610978254995717215-XXXXXXX-XX
sessionId: 2D7E2416131887D473F6CFD7B35769C
version: 13.7
@timestamp: 2022-12-26 11:13:40.657
typeError: ERROR
functionality: com.company.modules.factory.Factory
message: No CAB matches reading 'Invalid'
detail: class com.company.exceptions.MyException:
    at com.company.LancAdapter.do(LancAdapter.java:449)
    at com.company.CABWrapper.read(CABWrapper.java:191)
    ...
    at com.company.Main(Main.java:94)
user message : I got this error while I was trying to ...
```

Figure 1: A crash report with a stack trace and its context.

3.2 Approximate Nearest Neighbors Search

A similarity measure between two stack traces is a function denoted as $sim : \mathcal{D} \times \mathcal{D} \longrightarrow [0, 1]$. It can be any conventional similarity metric (Jaccard coefficient) or a specialized stack-trace similarity measure (e.g., PDM [8] and TraceSim [38]). The distance function is naturally given by $dist : \mathcal{D} \times \mathcal{D} \longrightarrow [0, 1]$ where $dist = 1 - sim$. Given a dataset $\mathcal{D}$ of N stack traces, the problem of nearest neighbor search under a user-defined similarity measure sim consists in finding, for a specific stack trace $s \in \mathcal{D}$, another stack trace denoted as $nn(s) \in \mathcal{D} \setminus \{s\}$ such that: $nn(s) = \arg\max_{s' \in \mathcal{D} \setminus \{s\}} sim(s, s')$.

An alternative of nearest neighbor search is the fixed-radius nearest neighbor (R−near neighbor) problem which seeks to find a set of stack traces $\mathcal{S}_R$ that are within the distance R of s ($0 < R < 1$), such that: $\mathcal{S}_R = \{s' \in \mathcal{D} \setminus \{s\} \mid dist(s, s') \leq R\}$.

There exists simple tree-based algorithms for approximate nearest neighbor search problems, notably KD trees [2] and SR-tree [18]. However, for large scale high-dimensional cases, these techniques

suffer from the well-known *curse of dimensionality* [3] where the performance is often surpassed by a linear scan. Consequently, significant research efforts have been dedicated to exploring highly efficient and scalable methods for approximating nearest neighbor search problems in large-scale datasets, including hashing and LSH.

Locality-Sensitive Hashing (LSH) has been particularly proposed to tackle the problem of *randomized or probabilistic approximate nearest neighbors search* [39], that is, targeting the ANN problem with guarantees aiming to find approximate nearest neighbors with probability rather than a deterministic way (which is not tractable). This choice is driven by the purpose of ensuring guarantees on the search accuracy with respect to the exact nearest neighbors search, while giving the user the ability to balance between precision and recall to a desired level. Formally, we define our problem as follows:

Problem 1 (Randomized approximate nearest neighbors search (RANN))**.** Given a new reported stack trace s, a dataset $\mathcal{D}$ of historical stack traces, the goal is to report some of the $R-$nearest neighbors $\mathcal{R}$ of s such that: $\mathcal{R} = \{s' \in \mathcal{D} \setminus \{s\} \mid Pr[s' \in \mathcal{S}_R] \geq 1 - \delta\}$ with $(0 < \delta < 1)$. The lower the parameter δ, the lower the chance of finding elements in the radius (i.e., more restrictive).

3.3 Hashing approach for the RANN problem

Hashing-based approaches attempt to map data features from the input space into a lower-dimensional space using hash functions so that the approximate nearest neighbors search on the resulting hash vectors can be performed efficiently. The compact hash codes generally belong to the Hamming space i.e., binary codes. We define the hash function for a stack trace s as $y = h(s)$ where y is the hash code and $h : \mathcal{D} \longrightarrow \{0,1\}^b$ where $b \geq 1$ is the number of bits in the hash code. In approximate nearest neighbors search settings, we usually opt for multiple hash functions to compute the final meta-hash code: $Y = H(s)$, where $H(s) = [h_1(s), h_2(s), ..., h_K(s)]^T$ and K is the number of hash functions. Hashing-based nearest neighbors search includes hash table lookup strategy [39] which seeks to design an efficient search scheme rooted in hash tables. The hash table is a data structure made of buckets, each of which is indexed by a meta-hash code such that the probability of collision of near-duplicate stack traces under a given similarity measure is maximized. Given a stack trace s, the stack traces $\{s' \in \mathcal{D} \setminus \{s\} \mid H(s) = H(s')\}$ are retrieved as near-duplicates of s. In order to improve the recall, we generally construct L hash tables containing hash buckets, each corresponding to a hash code $\{H_1, H_2, ..., H_L\}$. The near-duplicate stack traces are then defined as $\{s' \in \mathcal{D} \setminus \{s\} \mid \exists j \in [[1, L]], H_j(s) = H_j(s')\}$.

3.4 LSH for RANN problem

To address the problem 1 of randomized nearest neighbors search, Locality-Sensitive Hashing (LSH) [16] maps high dimensional data to lower dimensional representations by using a family $\mathcal{H}$ of random hash functions that satisfy the locality-sensitive property. Thus, similar data items in the high-dimensional input space are expected to have more chance to be mapped to the same hash buckets than dissimilar items. These similar data items are said to collide. Starting with a formal definition of an LSH family $\mathcal{H}$ to address our problem, we consider a metric space such that, $\mathcal{M} = (\mathcal{D}, dist)$, a

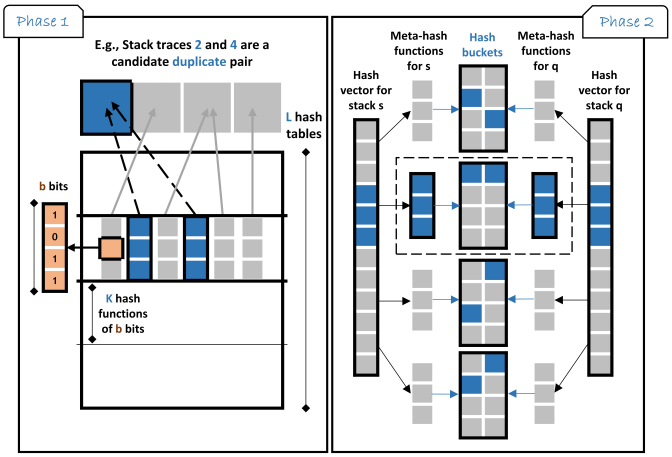

Figure 2: $(L, K)-$ parameterized LSH algorithm for retrieving near-duplicate stack traces with guarantees.

threshold $0 < R < 1$, an approximation factor $c > 1$, and two probabilities p_1 and p_2. The hash family $\mathcal{H}$ is a set of M hash functions $\{h_1, h_2, ..., h_M\}$ where each $h \in \mathcal{H}$ is defined as $h : \mathcal{D} \longrightarrow \{0,1\}^b$. An LSH family must satisfy the following conditions for any two stack traces $s, s' \in \mathcal{D}$ and any random hash function $h \in \mathcal{H}$:

- if $dist(s, s') \leq R$, then $Pr[h(s) = h(s')] \geq p_1$,
- if $dist(s, s') \geq cR$, then $Pr[h(s) = h(s')] \leq p_2$.

A family $\mathcal{H}$ is said to be $(R, cR, p_1, p_2)-$sensitive if $p_1 > p_2$. Alternatively [6], a sufficient condition for $\mathcal{H}$ to be an LSH family is that the *collision probability* should be monotonically increasing with the similarity i.e.,

$$Pr[h(s) = h(s')] = g(sim(s, s')), \tag{1}$$

where g is a monotonically increasing function. Indeed, most of popular known LSH families such as Minhash [4] for Jaccard similarity, satisfy this strong property.

The LSH scheme indexes all stack traces in hash tables and searches for near-duplicates via a hash table lookup strategy. The LSH algorithm uses two key hyperparameters L and K to be tuned. Given the LSH family $\mathcal{H}$, the LSH algorithm amplifies the gap between the high probability p_1 and the low probability p_2 by concatenating K hash functions chosen independently and uniformly at random from $\mathcal{H}$, to form a meta-hash function $H(s) = [h_1(s), h_2(s), ..., h_K(s)]^T$. The meta-hash function is associated with a bucket ID in a hash table. Intuitively, it reduces the chances of collision between similar stack traces, since this requires them to have the same value for each of the K hash functions (i.e., high precision over the recall). To improve the recall, L meta-hash functions $H_1, H_2, ..., H_L$ are sampled independently, each of which corresponds to a hash table. These meta-hash functions are used to map each stack trace into L hash codes, and L hash tables are constructed to index the corresponding buckets, each using K random hash functions. The LSH algorithm is conducted in two phases as illustrated in Fig. 2 (considering, $L = 4$ hash tables, for each we have $K = 3$ hash functions of $b = 4$ bits.)

Pre-processing phase: The L hash tables are built from N stack traces. A hash table is indexed by K hash functions constituting its meta-hash function. We store pointers to stack traces in hash tables, since storing them in the original format is memory intensive.

Querying phase: Given a stack trace s, the algorithm iterates over the L meta-hash functions in order to retrieve all stack traces that are hashed into the same bucket as s, then reports the union from all these buckets $\bigcup_{j=1}^{L}\{s' \in \mathcal{D} \mid H_j(s) = H_j(s')\}$. A $(L, K)-$parameterized LSH algorithm succeeds in finding candidate near-duplicates for a stack trace s with a sampling probability at least $1 - (1 - p^K)^L$, where p is the collision probability of LSH function. This means that $\delta = (1 - p^K)^L$ as defined in the problem 1 of randomized ANN. If property (1) holds, in particular for the identity function, i.e., $g(x) = I_x$, we can rely on the so-called *probability-similarity* relation between two different stack traces $s, s' \in \mathcal{D}$ such that:

$$P_{K,L}(s, s') = 1 - (1 - sim(s, s')^K)^L \tag{2}$$

4 DEEPLSH DESIGN METHODOLOGY

In order to address Problem 1 to efficiently retrieve near-duplicates, it is crucial to define suitable Locality-Sensitive Hashing (LSH) families for stack trace-based similarity measures. While several LSH families have been proposed for various similarity measures [4, 6, 9], there is currently no generic mechanism available to generate a family of hash functions that satisfies the locality-sensitive property for any user-defined similarity measure, especially non-linear measures that often require human expertise. To overcome this challenge, we introduce DeepLSH, a generic approach that solely requires the stack trace dataset and any user-defined similarity measure (measure-agnostic) as input. DeepLSH provides a family of hash functions that converges to the locality-sensitive property.

4.1 Learning a family of LSH functions

We exploit a deep supervised Siamese neural network with an original objective loss function to learn hash functions that converge to the locality-sensitive property for a given similarity measure. Fig. 3 shows the structure of the proposed model that combines two identical neural networks sharing the same structure and the same parameters Θ. As input, we provide the model with the set $\mathcal{G}$ of all possible distinct pairs of stack traces encoded as an ordered sequence of stack frames. Each distinct frame is then referred to as a feature. The model output is provided with the similarity values for each pair of stack traces. The model F, with its corresponding parameters Θ, consists in encoding a stack trace into a compact vector that represents a family of M concatenated binary hash codes, each of which is encoded in b bits in the Hamming space, denoted as $F_\Theta(s)$. The model consists in a concatenation of stacked convolution layers with different kernel sizes. We depict three kernel region sizes: 2, 3 and 4, each of which has 256, 512 or 1024 filters. These filters perform convolutions on the one-hot encoded stack frames to generate feature maps. Then, 1-Max pooling is performed over each map to record the largest number from each feature. Finally, the resulting features are concatenated to form a feature encoding vector for the penultimate layer that is fully connected to the hash model. It is noteworthy that any feature encoder structure (e.g., CNN, CNN-LSTM, AE, etc.) can be also used as a stack trace encoder instead of our proposed network architecture.

Given the two hash vectors of the Siamese neural network, our contribution consists in designing an objective loss function that efficiently conducts F_Θ to learn a family of binary hash functions that aim to converge to the locality-sensitive property for the given similarity function *sim*. We propose to leverage Property (1) which is sufficient to imply the two required conditions of an LSH family. Assuming that the function g is the identity function: $g(x) = I_x$ since the similarity values are within the closed interval $[0, 1]$, the set of parameters Θ are optimized such that the probability of two random projected hash functions of order k from the resulting hash vectors, h_k^i and h_k^j being equal, converges to the similarity value between the two stack traces s_i and s_j i.e.,

$$Pr[h_k^i = h_k^j] = Pr[H(s_i)_k = H(s_j)_k] = sim(s_i, s_j) \tag{3}$$

More formally, we seek to minimize the Mean Squared Error (MSE) between the probability of collision of two randomly projected hash functions of order k, i.e. h_k^i resp. h_k^j of $H(s_i)$ resp. $H(s_j)$, and the similarity value $sim(s_i, s_j)$, that is:

$$\arg\min_{\Theta} \sum_{(s_i,s_j)\in\mathcal{G}} \frac{1}{|\mathcal{G}|}[Pr[F_\Theta(s_i)_k = F_\Theta(s_j)_k] - sim(s_i, s_j)]^2 \tag{4}$$

At this point, the challenge is to formalize the probability of collision in the loss function. In other words, we attempt to quantify the probability $Pr[F_\Theta(s_i)_k = F_\Theta(s_j)_k]$ during the learning phase. In the following, we present the complete procedure for designing a computed loss function for the model F.

4.2 Objective loss function

Given that the hash vectors are in a Hamming space, i.e., the vectors are restricted to binary values, $\{0, 1\}$ or $\{-1, 1\}$, it can be demonstrated that calculating the collision probability between two randomly projected hash functions of the same order h_k^i and h_k^j is equivalent to computing the Hamming similarity between the two hash vectors $H(s_i)$ and $H(s_j)$. This equivalence is satisfied since, for the Hamming similarity when $b = 1$, it has been proven in [16], that the projection function (i.e., a single bit drawn randomly) verifies the locality-sensitive property. In other terms, for two binary vectors x and x' of length d with a Hamming distance r, the collision probability by randomly pulling a hash function from the set $\{h : [-1, 1]^d \rightarrow \{-1, 1\} \mid h(x) = x_i, i \in \{1, ..., d\}\}$ verifies:

$$Pr[h(x) = h(x')] = g(r) = 1 - \frac{r}{d} \tag{5}$$

Intending to leverage property (5) and given that our hash functions are rather b-bit encoded ($b \geq 1$), i.e., not restricted to a single projection but a succession of b coordinates, we need to generalize this property for $b \geq 1$. Consequently, we define a generalized Hamming distance between two hash vectors $H(s_i)$ and $H(s_j)$ as the number of different projected hash functions of order k: $|\{k \in [[1, M]] \mid h_k^i \neq h_k^j\}|$. As a result, each hash function that belongs to $\{h' : [-1, 1]^{M\times b} \rightarrow \{-1, 1\}^b \mid h'(s) = H(s)_k\}$ satisfies the locality-sensitive property. This leads to conclude that for two different stack traces s_i and s_j, the collision probability between two projected hash functions of a specific order k' referring to (5):

$$Pr[h_{k'}^i = h_{k'}^j] = 1 - \frac{|\{k \in [[1, M]] \mid h_k^i \neq h_k^j\}|}{M} \tag{6}$$

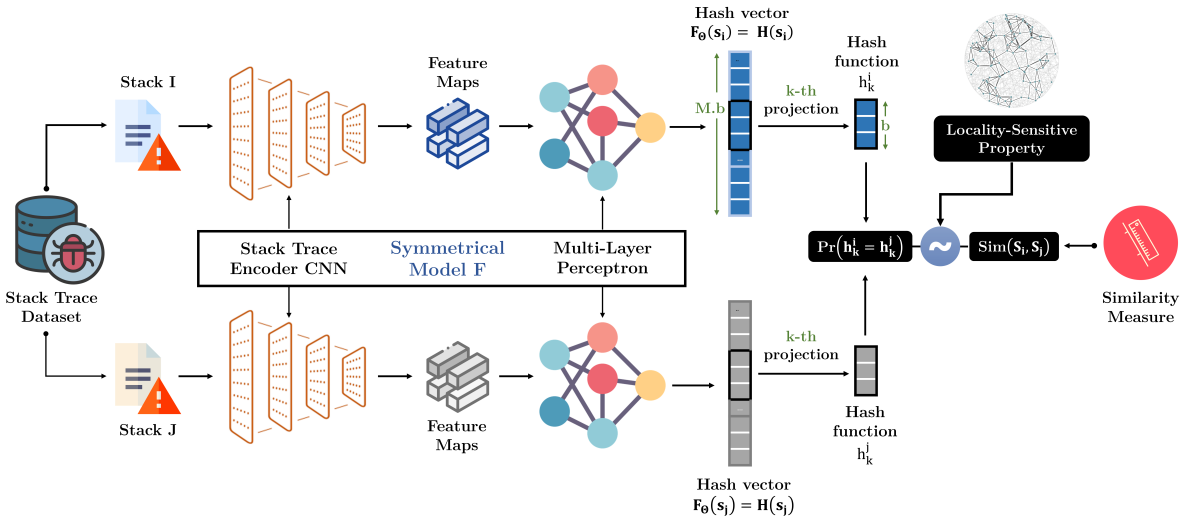

Figure 3: DeepLSH: Deep Siamese hash learning neural network overview

Correspondingly, referring to the property (6), the objective function as described in (4) can be formalized as follows:

$$\sum_{(s_i,s_j)\in\mathcal{G}} \frac{1}{|\mathcal{G}|}[1 - \frac{|\{k \mid h^i_k \neq h^j_k, k \in [[1, M]]\}|}{M} - sim(s_i, s_j)]^2 \quad (7)$$

A challenging problem in hashing, on the other hand consists in dealing with the binary constraint on hash vectors. This binary constraint leads to NP-hard mixed integer optimization problem [11]. In particular, the challenge in neural network parameter optimization is the *vanishing gradient descent* from the `Sign` function used to obtain binary values. Specifically, the gradient of the `Sign` function is zero for all non-zero input values, which is limiting for neural networks that rely on gradient descent for training. In order to handle this challenge, most deep hashing techniques relax the constraint during the learning of hash functions using `Sigmoid` or `Hyperbolic Tangent` functions [13, 14, 21, 26]. With this relaxation, the continuous hash codes are learned first. Then, the codes are binarized with thresholding. Continuous relaxation is a simple approach to address the original binary constraint problem. However, with binary hash codes that result from thresholding in the test phase, the solution may be suboptimal, compared to including the binary constraint in the learning phase.

To this extent, we propose a simple yet efficient solution to cope with the binary constraint in the training phase. The solution lies in using approximate Hamming similarity. It requires having continuous values that are extremely close to binary values $\{-1, 1\}$. We propose to use the `Hyperbolic Tangent` activation on the hash layer while including the following condition in the loss function to drive the absolute hash values to be exceedingly close to 1:

$$\frac{1}{M \cdot b} H(s)^T . H(s) - 1 = 0. \quad (8)$$

Under this regularization term incorporated into the loss function, we define the approximate generalized Hamming similarity as follows:

$$gHam\left(H(s_i), H(s_j)\right) = 1 - \frac{\sum_{k=1}^{M} D_{\text{Chebyshev}}(h^i_k, h^j_k)}{2 \cdot M}, \quad (9)$$

where $D_{\text{Chebyshev}} = \max_{l\in\{1,\ldots,b\}}(|h^i_{k,l} - h^j_{k,l}|)$

The Chebyshev distance between $h^i_{k,l}$ and $h^j_{k,l}$ is then given as the maximum absolute distance in one of the b dimensions. This implies that two hash codes are assumed to be similar if all bits of the hash code are matched for a specific projection. In other words, if $\exists\, l \in \{1, \ldots, b\}$ for a specific k such that $|h^i_{k,l} - h^j_{k,l}| \approx 2$, then h^i_k and h^j_k are considered as two different hash codes.

Finally, to ensure independence between the hash code bits along with the load-balanced locality-sensitive hashing, and inspired by the work of [11], we have introduced the following regularization term that pushes the model to diversify the hash codes:

$$\frac{1}{M \cdot b} H(s)^T . \mathbb{1}_{M \cdot b} = 0. \quad (10)$$

Putting all together. Having all the necessary elements to design an appropriate objective loss function to be optimized for DeepLSH model, we define for convenience the following notations. Let $S = \{sim(s_i, s_j)\}_{i,j\in[[1,N]]} \in [0, 1]^{N\times N}$ be the matrix representation of the similarities between all the stack trace pairs, and $\mathcal{H} = [H(s_1), H(s_2), \ldots, H(s_N)] \in [-1, 1]^{M\cdot b\times N}$ be the approximate binary hash vectors generated by the model F_Θ, such that, $H(s_i) = [h^i_1, h^i_2, \ldots, h^i_M]^T \in [-1, 1]^{M\cdot b}$. We refer to $\mathcal{W} = \{gHam(H(s_i), H(s_j))\}_{i,j\in[[1,N]]} \in [0, 1]^{N\times N}$ as the matrix representation of the generalized Hamming similarity between all pairs of hash vectors produced from the model F_Θ. We formulate the following optimization problem to learn the parameters of our

DeepLSH model using gradient descent as follows:

$$\begin{aligned}\min_{\Theta} \mathcal{L}_{\text{DeepLSH}} = & \frac{1}{|\mathcal{G}|}\|\mathcal{W} - S\|^2 \\ & + \frac{\lambda_1}{2}\|\frac{1}{M \cdot b}\mathcal{H}^T\mathcal{H} - \mathbf{I}_N\|^2 \\ & + \frac{\lambda_2}{|\mathcal{G}|}\|\frac{1}{M \cdot b}\mathcal{H}^T\mathbb{1}_{M \cdot b}\|^2 \\ & + \lambda_3\|\Theta\|_F^2, \end{aligned} \tag{11}$$

where λ_1, λ_2 and λ_3 are regularization parameters to assess the importance of the different parts of the objective function.

5 EXPERIMENTS

5.1 Experimental Setup and Evaluation Protocol

We report our experimental study to assess the effectiveness of DeepLSH in performing efficient, fast, and scalable approximate nearest neighbors search, by providing appropriate hash functions that can approximate the stack trace similarity measures and allow them to scale when used in large databases of bug reports.

Stack trace dataset and training methodology. Our experiments are conducted on a real-world dataset comprising stack traces automatically reported by our ERP software. To establish a robust training dataset, we selectively choose the most frequent stack traces from our historical incident database, creating distinct pairs of stack traces. These pairs are then utilized to train our DeepLSH model. Each pair is assigned a similarity value calculated using diverse similarity functions. We evaluate the performance of DeepLSH on twelve different similarity measures: Jaccard (bag-of-words and bi-grams), Cosine (bag-of-words, bi-grams, and TF-IDF), Edit distance [1], PDM [8], Brodie [5], DURFEX [33], Lerch [22], Moroo [30], and TraceSim [38]. It is important to note that these similarity metrics serve as a reference point and act as the ground truth in our current setup. Our objective, therefore, is not to evaluate the effectiveness of these similarity measures or compare them with each other. This is because each measure is applied to a vast dataset of stack traces, where labeled information is not always available, or manual labeling is impractical, especially in cases of frequent background process failures that may occur in the thousands per day. Our goal is to ensure that regardless of the used similarity measure employed for stack traces, DeepLSH approach can effectively replicate this measure of similarity. Additionally, it should be capable of scaling up and being utilized within large-scale systems.

Regarding the training methodology, the training set consists of 499500 pairs of stack traces, while the validation and test set are constituted of 99900 pairs. The number of hash functions M and the size of each hash code b can be parameterized by the user. By default these values are respectively set to 64 hash functions of 8 bits. The max iteration is fixed at 20 epochs with a batch size of 256 or 512. The parameter optimization process is achieved with the readily available `Adam` optimizer of `TensorFlow` with an adaptive learning rate and a weight decay of $1e^{-4}$. With this configuration, the training process takes barely 10 minutes. The source code and data with all associated instructions required for experimental replication are made available [*]

[*] https://github.com/RemilYoucef/deep-locality-sensitive-hashing

Baselines. As there is no explicit competing approach for DeepLSH in the state-of-the-art, we chose to initially compare with **(1) Standard LSH methods**, namely Min-Hash and Sim-Hash. This comparison should only be performed with the Jaccard and Cosine metrics respectively since they are not generalizable to other measures compared to DeepLSH. As mentioned in Sec 2, we compare against the closest method to our work referred to as **(2)** `CNNH+LSH` [14]. This approach uses the concept of learn to hash and then performs in a post-processing step the Hamming LSH. The methodology followed in this work is significantly different to ours. DeepLSH unlike the latter incorporates the LSH component into the model learning phase, resulting in a new loss function with related regularization to meet the locality-sensitive property. Regarding the application of [14] on stack-traces, since it was primarily designed for images, we only needed to provide a list of one-hot encoded stack-frames to the convolutional feature extractor instead of pixels. Finally, to evaluate the scalability of our approach, we compare against **(3) Native k-NN (k-Nearest neighbors)** approach of linear complexity, using the exact computation of similarity functions between stack traces. It is noteworthy that clustering techniques have not been considered as baselines (as discussed in the introduction), since the addressed problem is an ANN search.

Evaluation protocol. Through this experimental study, we address the following research questions by proposing an evaluation protocol to assess each point claimed in this work:
RQ1 [Model Evaluation]: Does DeepLSH model manage to converge to the locality-sensitive property to mimic a diverse set of stack-trace-based similarity metrics? We first highlight, by means of the Kendall τ ranking coefficient [19], whether the model succeeds in preserving the original order between the predicted pairwise similarities. In addition, we study how accurately the generalized Hamming similarity approximates the true Hamming similarity between the discretized hash vectors in the test phase.
RQ2 [DeepLSH for ANN search]: Does DeepLSH model achieve satisfactory performance in finding near-duplicate crash reports using a given similarity measure? By querying the model to obtain the hash vectors for unseen stack traces, we study the search performance to retrieve approximately near-duplicate stack traces based on two metrics that are widely used in the context of crash deduplication: Recall Rate at the first k positions (RR@k) [35] and the Mean Reciprocal Rank (MRR) [7].
RQ3 [Preserving LSH guarantees]: To what extent does DeepLSH succeed in preserving the guarantees of LSH compared to Standard LSH methods and the baseline (`CNNH+LSH`) [14]. For this purpose, we provide recall, precision and F-score measures adjusted to quantify the extent to which the probability-similarity constraint (2) has been satisfied (more details are provided hereafter).
RQ4 [Runtime Analysis]: How does the scalability of DeepLSH compare to that of a native linear k-NN approach? We report the execution time required for DeepLSH to find the near-duplicates, compared to a k-NN approach of linear time complexity.

5.2 RQ1: Model Evaluation

In Fig. 4, we highlight the strong linear correlation between the probability of hash collision and the similarity values for almost

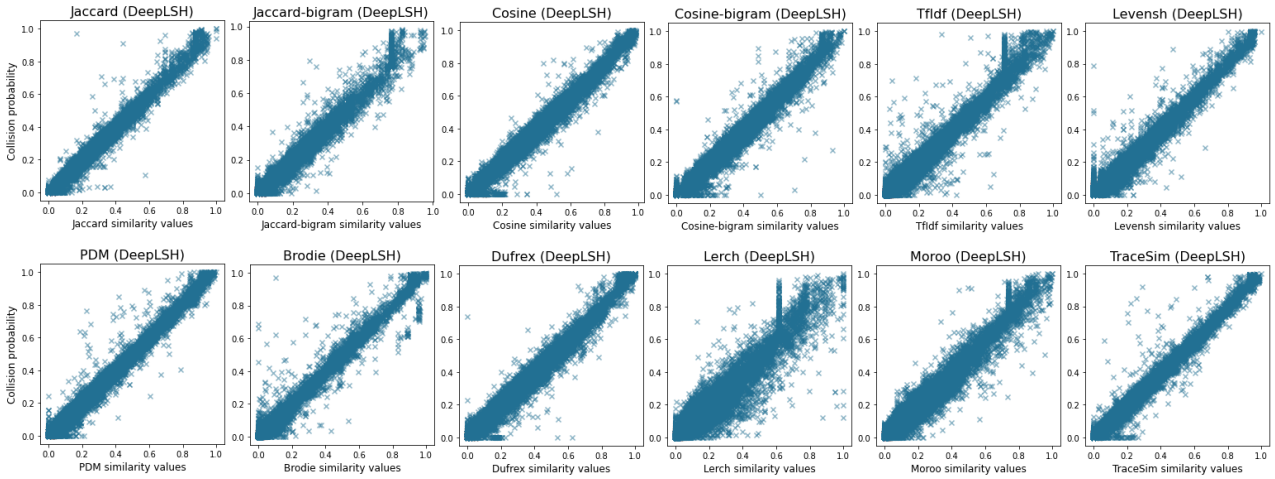

Figure 4: Locality-sensitive preserving: Correlation between the probability of hash collision and the similarity value

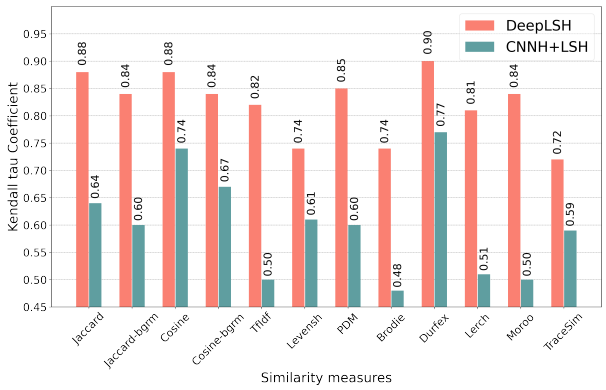

Figure 5: Kendall's τ coefficient between the real and predicted pairwise similarities

all similarity measures performed on stack trace pairs. The resulting plots show that the model is able to converge perfectly to the locality-sensitive property for almost all similarity measures. We observed a few outliers in the TF-IDF, Lerch, and Moroo measures. These outliers can be attributed to extremely low or high IDF values, indicating frames that are either non-discriminatory or infrequent among stack traces. Fortunately, their presence does not significantly affect the model's performance. However, to further address this issue, we can consider augmenting the feature set by including the IDF of the frame features from the training set. It is also important to assess the capability of DeepLSH model to maintain the order between pairwise similarity values. For instance, for a triplet of stack traces s, p and q if $sim(s, p) > sim(s, q)$, we aim to evaluate whether the model is likely to provide hash functions s.t. $Pr[h_k(s) = h_k(p)] > Pr[h_k(s) = h_k(q)]$ for $k \in M$. For this purpose, we measure the Kendall rank correlation coefficient, between the set of similarity values, and the set of generalized Hamming similarities between the resulting hash vectors as shown in Fig. 5.

Remarkably, we obtained satisfactory results compared to our baseline (on average, 0.82 for DeepLSH, against 0.60 for CNNH+LSH, that is, 0.22 of improvement) which permitted to achieve better and accurate results on the ANN search. Finally, thanks to the regularization conditions (8) and (10) incorporated in the objective loss function, the model yields approximate binary hash values extremely close to $\{-1, 1\}$ that are binarized/relaxed in the test phase. We seek to evaluate whether our proposed solution to deal with the binarization problem using the generalized Hamming similarity (9) performed in the training phase, is optimal and captures the true Hamming similarity between the discritized hash values in the test phase. Considering the TraceSim measure as an example in Fig. 6 (on the left), we notice a strong linear correlation between the true hamming similarity calculated on the binary vectors and the approximate generalized Hamming similarity used in the loss function during the learning phase. This means that our loss optimization process is as identical as the optimization of any loss function with strictly binary values in the training phase. In the same Figure (on the right), we show the impact of not incorporating the LSH component into the model, as has been done in [14]. Performing LSH on the discretized vectors in a post-processing step results in a sub-optimal optimisation, since the correlation between the similarity calculated using basic embedding and the true Hamming similarity is not even monotonic. Consequently, CNNH+LSH has failed to capture TraceSim similarity.

5.3 RQ2: Evaluation of DeepLSH for ANN

The objective of our DeepLSH approach, given a similarity measure, is to generate, for a new stack trace s reported by our monitoring system, an appropriate hash vector to query a $(L, K)-$parameterized LSH for quickly and efficiently locate in a sub-linear time complexity its near-duplicates. The hash vector contains M hash functions, partitioned across L hash tables, each consisting of a concatenation of K hash functions called a meta-hash function of size $K \cdot b$. A stack trace $q \in \mathcal{R}_s$ is identified as a near-duplicate stack of s if it matches the stack trace s at least in one meta-hash function. The

Table 1: Comparison between the search performances of DeepLSH against the standard LSH approaches w.r.t. their addressed similarity measures and (CNNH+LSH) [14] in terms of Recall Rate (RR@k) and Mean Reciprocal Rank (MRR).

Similarity Measure	RR@1				RR@5				MRR			
	CNNH+LSH	DeepLSH	MinHash	SimHash	CNNH+LSH	DeepLSH	MinHash	SimHash	CNNH+LSH	DeepLSH	MinHash	SimHash
Jaccard	0.71	0.87	**0.90**	–	0.79	**0.92**	**0.92**	–	0.85	**0.96**	0.95	–
Jaccard-bigram	0.67	0.87	**0.90**	–	0.76	0.91	**0.92**	–	0.82	0.93	**0.94**	–
Cosine	**0.84**	0.81	–	0.61	0.83	**0.90**	–	0.62	**0.88**	0.87	–	0.80
Cosine-bigram	0.76	**0.84**	–	0.58	0.79	**0.93**	–	0.58	0.89	**0.91**	–	0.80
TF-IDF	0.73	**0.76**	–	0.55	0.75	**0.88**	–	0.55	0.85	**0.90**	–	0.73
Edit Distance [1]	0.81	**0.88**	–	–	0.75	**0.94**	–	–	0.88	**0.95**	–	–
PDM [8]	0.80	**0.84**	–	–	0.76	**0.90**	–	–	0.82	**0.93**	–	–
Brodie [5]	0.79	**0.84**	–	–	0.76	**0.90**	–	–	0.82	**0.93**	–	–
DURFEX [33]	0.72	**0.83**	–	–	0.79	**0.91**	–	–	0.82	**0.91**	–	–
Lerch [22]	0.70	**0.78**	–	–	0.70	**0.85**	–	–	0.80	**0.88**	–	–
Moroo [30]	0.75	**0.80**	–	–	0.68	**0.90**	–	–	0.80	**0.93**	–	–
TraceSim [38]	**0.81**	0.79	–	–	0.75	**0.90**	–	–	0.84	**0.92**	–	–

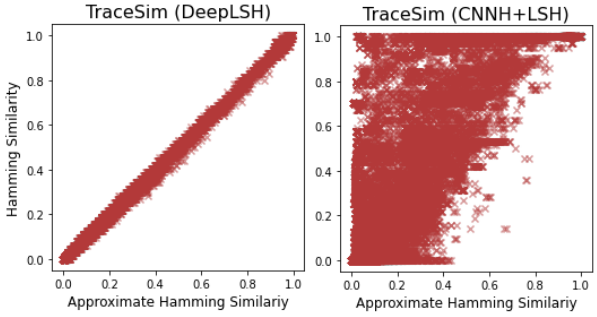

Figure 6: Comparison between DeepLSH and [14] in preserving the Hamming similarity between hash vectors

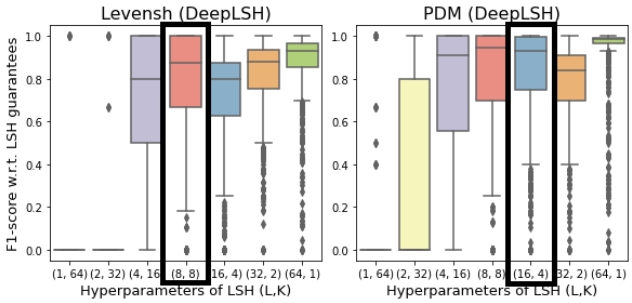

Figure 7: F-score boxplots w.r.t. different values of (L,K)

set of near-duplicate stack traces of s is denoted by $\mathcal{R}_s$. It is worth noting that the set $\mathcal{R}_s$ is sorted in the original order with respect to the similarity measure value.

LSH offers the possibility to control the trade-off between precision and recall (w.r.t LSH guarantees) by setting the hyperparameters values K and L. We can simply choose to consider the tuple of hyperparameters that maximizes the F-score. However, as shown in Fig. 7 (e.g., $(L, K) = (16, 4)$ for PDM), we ignore extreme cases i.e., cases where the threshold is very small or very large, which refers to a very large value of L or K, or cases where the variance is very high (for more details on how the F-score is calculated, refer to 5.4). As an example, when using the combination $(L, K) = (64, 1)$, we observe higher F-score values. However, it's important to note that this is partly a result of selecting a very low threshold, which in turn leads to a larger number of near-duplicates that need to be analyzed.

In a first analysis, we were interested in the *recall rate of order* k. For each stack trace s that belongs to a query set Q, we yield a set of its approximate nearest neighbors $\mathcal{R}_s$ (i.e., potential near-duplicates) such that $|\mathcal{R}_s| \geq k$ and hence,

$$\textbf{RR@k} = \frac{1}{k \cdot |Q|} \sum_{s \in Q} \sum_{i=1}^{k} \mathbb{1}_{[nn_i(s, ^*\text{args}) \in \mathcal{R}_s]},$$

where $nn_i(s, ^*\text{args})$ is a function that returns the real nearest neighbor of order i for the stack trace s given a set of historical stack traces and the LSH hyperparameters L and K.

In order to evaluate the ranking quality of a set of near-duplicates $\mathcal{R}_s$ for a stack trace s according to a (L, K) combination, and relative to the set of true nearest neighbors $\mathcal{T}_s$, we use the *Mean reciprocal rank* (MRR) [7]. This measure seeks to compute the reciprocal rank of a retrieved near-duplicate $s' \in \mathcal{R}_s$ relative to its actual position in the set of true nearest neighbors. More concretely:

$$\textbf{MRR} = \frac{1}{|Q|} \sum_{s \in Q} \frac{1}{|\mathcal{R}_s|} \sum_{s' \in \mathcal{R}_s} \frac{rank(s', \mathcal{R}_s)}{rank(s', \mathcal{T}_s)}$$

E.g., let's consider a given stack trace q, where we retrieve the set of its approximate nearest neighbors and subsequently sort them according to the original order $\mathcal{R}_q = \{s_2, s_3, s_5\}$ and the set of its true nearest neighbor is given as $\mathcal{T}_q = \{s_1, s_2, s_3, s_4, s_5\}$. The MRR is then: $\frac{1}{3}(\frac{1}{2} + \frac{2}{3} + \frac{3}{5}) \approx 0.63$. The MRR in this case is over penalized since we failed to find the true nearest neighbor s_1 in $\mathcal{R}_q$. It is noteworthy that the set $\mathcal{R}_q$ is sorted according to the original order w.r.t to the similarity measure.

The results are presented in detail in Table 1, according to the identified similarity measures that have been proposed to address the crash-deduplication problem. We choose 2 different values of $k = \{1, 5\}$ for the recall rate. We compare DeepLSH against MinHash, SimHash and CNNH+LSH. Ideally, standard LSH techniques

Table 2: Comparison between the precision/recall and f-score of DeepLSH in preserving the probability-similarity relation (2) against the standard LSH approaches w.r.t. theirs addressed similarity measures and (CNNH+LSH) [14].

Similarity Measure	Precision				Recall				F-score			
	CNNH+LSH	DeepLSH	MinHash	SimHash	CNNH+LSH	DeepLSH	MinHash	SimHash	CNNH+LSH	DeepLSH	MinHash	SimHash
Jaccard	0.64	**0.78**	0.76	–	0.78	**0.85**	**0.85**	–	0.70	**0.81**	0.80	–
Jaccard-bigram	0.56	**0.76**	0.70	–	0.70	0.74	**0.83**	–	0.62	**0.75**	**0.75**	–
Cosine	**0.77**	0.72	–	0.74	0.74	**0.84**	–	0.6	0.75	**0.78**	–	0.66
Cosine-bigram	**0.74**	**0.74**	–	0.66	0.72	**0.82**	–	0.41	0.73	**0.78**	–	0.50
TF-IDF	**0.85**	0.76	–	0.49	0.61	**0.86**	–	0.62	0.71	**0.81**	–	0.55
Edit Distance [1]	0.37	**0.78**	–	–	0.78	**0.88**	–	–	0.50	**0.83**	–	–
PDM [8]	0.68	**0.85**	–	–	0.76	**0.86**	–	–	0.72	**0.85**	–	–
Brodie [5]	0.36	**0.83**	–	–	0.81	**0.86**	–	–	0.50	**0.84**	–	–
DURFEX [33]	0.73	**0.78**	–	–	0.70	**0.79**	–	–	0.71	**0.78**	–	–
Lerch [22]	0.74	**0.76**	–	–	0.57	**0.76**	–	–	0.64	**0.76**	–	–
Moroo [30]	0.66	**0.73**	–	–	**0.85**	0.82	–	–	0.74	**0.77**	–	–
TraceSim [38]	0.31	**0.80**	–	–	0.84	**0.88**	–	–	0.45	**0.84**	–	–

should guarantee optimal search accuracy compared to DeepLSH, since they are proven to converge to the locality-sensitive property w.r.t. their similarity measures. Interestingly, we observe that DeepLSH almost matches the search performances of MinHash on Jaccard similarity, and outperforms SimHash. In fact, while MinHash is a reliable probabilistic model designed for estimating Jaccard similarity with guarantees, it lacks the versatility to be extended to other similarity metrics, especially those intended for comparing stack traces. This limitation significantly restricts its applicability to the crash deduplication problem. Jaccard measure, which it relies on, may not be the most suitable metric for stack trace comparison as it does not account for the order of frames or the sequential aspect of function invocation within stack traces. This demonstrates also that DeepLSH is not only generalizable to other complex measures, but can even be used for measures where an existing LSH is already known. We also show that DeepLSH outperforms CNNH+LSH with a large margin on 3 different comparison metrics and for almost all similarity measures. More specifically, we notice that DeepLSH search performance is enhanced with a larger value of k up to 0.94 for Edit distance with an improvement of $\sim$ 0.2 over CNNH+LSH.

5.4 RQ3: LSH guarantees Preserving

In what follows, we aim to evaluate whether DeepLSH succeeds in preserving the guarantees of LSH regarding the probability-similarity relation in (2). To this end, for a specific stack trace s we look for the true near-duplicate stack traces $q \in \mathcal{R}_s^{\text{True}}$ that the model should return with a (K, L) setting, s.t. $P_{K,L}(s, q) = 1 - (1 - sim(s, q)^K)^L \geq 0.5$, i.e., the probability to belong to the set is equal or higher than 0.5. We then derive the precision, recall and F-score between the returned set of near-duplicates $\mathcal{R}_s$ and $\mathcal{R}_s^{\text{True}}$. More formally we derive this values, s.t:

$$\textbf{Recall} = \frac{1}{|Q|} \sum_{s \in Q} \frac{|\mathcal{R}_s \cap \mathcal{R}_s^{\text{True}}|}{|\mathcal{R}_s^{\text{True}}|}$$

$$\textbf{Precision} = \frac{1}{|Q|} \sum_{s \in Q} \frac{|\mathcal{R}_s \cap \mathcal{R}_s^{\text{True}}|}{|\mathcal{R}_s|}$$

Table 3: Comparison between the runtime required to find near-duplicate stack traces for DeepLSH, k-NN based approach and standard LSH techniques.

Similarity Measure	Runtime (~ Seconds)				
	k-NN	CNNH+LSH	DeepLSH	MinHash	SimHash
Jaccard	258	30	26	57	-
Cosine	8288	15	14	-	3
TF-IDF	8510	16	15	-	4
Edit Distance [1]	4911	29	29	-	-
PDM [8]	10047	16	16	-	-
Brodie [5]	Limit	27	27	-	-
DURFEX [33]	12160	26	24	-	-
Lerch [22]	3118	24	24	-	-
Moroo [30]	15253	25	25	-	-
TraceSim [38]	13050	30	30	-	-

$$\textbf{F-score} = \frac{2}{\text{Recall}^{-1} + \text{Precision}^{-1}}$$

As detailed in Table 2, one can notice that DeepLSH is much better than all baselines in terms of F-score on almost all similarity measures including the accurate Minhash for Jaccard. DeepLSH showed significantly better performance in terms of recall, i.e., its ability to capture all similarities that are beyond the threshold imposed by an optimal parameterization of (K, L). On the other hand, the reported precision values, as opposed to CNNH+LSH, show that DeepLSH does not generate false positives, so that false near-duplicates are not grouped in the same bucket. In particular, on similarity measures that use the Levenshtein distance (e.g. ED, Brodie and TraceSim), we observe a rather low precision for CNNH+LSH, which shows on the one hand the limitation of CNNH+LSH to generalize to such metrics, and on the other hand agrees with the explanation of Fig. 6.

5.5 RQ4: Runtime Analysis

We evaluate the scalability of DeepLSH and how quickly this approach identifies, for a batch of stack traces in a large historical crash database, the most similar stacks w.r.t. a given similarity measure. We report the execution time required to find the near-duplicate candidates for 1,000 new stack traces when querying

on more than 100,000 historical crashes (100 million queries). We compare basically against the native k-NN based approach for all similarity measures. Recall that uni-gram and big-gram representations yield identical results for the cosine, Jaccard, and TF-IDF measures. The reported results are depicted in Table 3. The execution time for a native k-NN approach depends on the batch size, the size of the database and the computational complexity of the similarity measure. With the latter, most similarity measures under the conditions described above require more than an hour to return any results (more than 3 hours for DURFEX, Moroo and TraceSim) and no results returned by Brodie within 10 hours. On the other hand, DeepLSH only depends on the number of hash tables which does not exceed 64 tables. We notice that the runtime is roughly constant and around $24 \sim 27$ seconds on average. Remarkably, DeepLSH is even faster than MinHash for Jaccard Similarity. SimHash, on the other hand, has proven to be faster, and `CNNH+LSH` has been similar to DeepLSH in runtime, but as seen above, both approaches show poor search performance and lower guarantees.

5.6 Experiments Discussion and Perspectives

Through our experiments, we have successfully demonstrated the effective utilization of DeepLSH w.r.t. a diverse range of similarity measures, specifically designed for comparing bug reports. By preserving the locality-sensitive hashing property, our approach offers reliable guarantees on the search accuracy of retrieving near-duplicates while significantly enhancing the retrieval time. When evaluated against baseline methods, DeepLSH exhibited highly satisfactory performance, successfully identifying the most similar stack traces based on the chosen similarity measure used as a reference point. Notably, DeepLSH is a groundbreaking similarity-agnostic approach that applies hashing techniques in this specific context, while matching well-known hashing technique such as MinHash and even surpassing SimHash which are tailored for only a single similarity measure. It may be questioned why we don't simply employ MinHash with the Jaccard coefficient, as it already delivers satisfactory performance and scalable results. However, the Jaccard coefficient is not the most appropriate measure for comparing stack traces. That is why various similarity measures have been proposed in the field of crash deduplication to address the unique characteristics of stack traces. Unfortunately, MinHash is not suitable for these alternative similarity measures and cannot be effectively applied. Furthermore, our runtime analysis has revealed that DeepLSH exhibits impressive computational efficiency, regardless of the complexity of the chosen similarity measure. This remarkable finding allows users to freely choose any similarity measure suitable for their system, ensuring both high search accuracy and fast retrieval. Moreover, our method does not only offer users the flexibility to adjust the trade-off between recall and precision through the values of K and L but also provides a systematic approach to selecting the optimal hyperparameters' combination that maximizes the F-score w.r.t. LSH property preserving, as explained in Figure 7. It's worth mentioning that augmenting the number of hash tables may slightly increase computation time but remains well within the order of a few seconds. Overall, the experimental results clearly demonstrate the superiority of DeepLSH in terms of both performance and flexibility and offer an innovative solution for fast near-duplicate stack traces retrieval.

As a future perspective, we intend to evaluate our approach on other large public datasets of stack traces, which might be available but not exclusively specified for Java stack traces. It is also noteworthy that this method can be easily adapted to various settings, applied to other datasets, and generalized to many other use cases. For instance, it could be utilized for text similarity tasks with complex metrics like Word Mover's Distance [15], in conjunction with natural language processing techniques. This area presents an exciting opportunity for exploration that we plan to delve into.

6 CONCLUSION

In this paper, we tackle the important task of fast and efficient automatic crash bucketing in software development. We investigate the potential of locality-sensitive hashing (LSH) for this purpose, leveraging its sublinear performance and theoretical guarantees in terms of accuracy for similarity search. This approach offers significant advantages when dealing with large datasets, yet surprisingly, LSH has not been explored in the crash bucketing literature. The main reason for the lack of consideration of LSH in crash bucketing research is the challenge of deriving hash functions that satisfy the locality-sensitive property for advanced and complex crash bucketing metrics. To address this gap, we propose a novel, parameterizable approach named DeepLSH. We introduce an original objective loss function, complemented by appropriate regularizations, enabling convergence to the desired locality-sensitive property. By doing so, DeepLSH can mimic any given similarity metric, thereby enhancing and improving the time and efficiency of near-duplicate crash report detection. Overall, our findings highlight the untapped potential of LSH in the crash bucketing domain. We present DeepLSH as a practical solution that effectively improves the time and efficiency of automatic crash bucketing. Furthermore, DeepLSH maintains compatibility with various similarity metrics, making it a versatile tool for software developers.

ACKNOWLEDGMENTS

This research received partial support from the ANR project 'Analogies: from theory to tools and applications' (AT2TA), ANR-22-CE23-0023. The authors extend their gratitude to Infologic Engineering for funding this research and express appreciation to Agence National Recherche Technologie (ANRT) for their financial support. Special thanks are extended to Professor Martin Montperrus for his insightful and valuable feedback, which significantly enhanced the quality of this paper.

REFERENCES

[1] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. 2008. Finding Similar Failures Using Callstack Similarity. In *SysML*.
[2] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.
[3] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When Is "Nearest Neighbor" Meaningful?. In *Database Theory - ICDT '99, 7th International Conference, 1999, Proceedings*. 217–235.
[4] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic Clustering of the Web. *Comput. Networks* 29, 8-13 (1997), 1157–1166.
[5] Mark Brodie, Sheng Ma, Guy M. Lohman, Laurent Mignet, Natwar Modani, Mark Wilding, Jon Champlin, and Peter Sohn. 2005. Quickly Finding Known Software

Problems via Automated Symptom Matching. In *Second International Conference on Autonomic Computing (ICAC 2005)*. IEEE Computer Society, 101–110.
[6] Moses Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing*. 380–388.
[7] Nick Craswell. 2009. *Mean Reciprocal Rank*. Springer US, 1703–1703.
[8] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *34th International Conference on Software Engineering, ICSE*. 1084–1093.
[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry*. 253–262.
[10] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *IEEE 27th International Conference on Software Maintenance, ICSM*. 333–342.
[11] Thanh-Toan Do, Anh-Dzung Doan, and Ngai-Man Cheung. 2016. Learning to Hash with Binary Deep Neural Network. In *Computer Vision - ECCV 2016 - 14th European Conference*. 219–234.
[12] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince R. Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP*. 103–116.
[13] Kun He, Fatih Çakir, Sarah Adel Bargal, and Stan Sclaroff. 2018. Hashing as Tie-Aware Learning to Rank. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 4023–4032.
[14] Weiming Hu, Yabo Fan, Junliang Xing, Liang Sun, Zhaoquan Cai, and Stephen J. Maybank. 2018. Deep Constrained Siamese Hash Coding Network and Load-Balanced Locality-Sensitive Hashing for Near Duplicate Image Detection. *IEEE Trans. Image Process.* 27, 9 (2018), 4452–4464.
[15] Gao Huang, Chuan Guo, Matt J. Kusner, Yu Sun, Fei Sha, and Kilian Q. Weinberger. 2016. Supervised Word Mover's Distance. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*. 4862–4870.
[16] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*. 604–613.
[17] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.
[18] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data*. 369–380.
[19] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.
[20] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. 486–493.
[21] Hanjiang Lai, Yan Pan, Ye Liu, and Shuicheng Yan. 2015. Simultaneous feature learning and hash coding with deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 3270–3278.
[22] Johannes Lerch and Mira Mezini. 2013. Finding Duplicates of Your Yet Unwritten Bug Report. In *17th European Conference on Software Maintenance and Reengineering, CSMR*. 69–78.
[23] Qi Li, Zhenan Sun, Ran He, and Tieniu Tan. 2017. Deep Supervised Discrete Hashing. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. 2482–2491.
[24] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. 2015. Deep learning of binary hash codes for fast image retrieval. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 27–35.
[25] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. 2015. Deep hashing for compact binary codes learning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2475–2483.
[26] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2016. Deep Supervised Hashing for Fast Image Retrieval. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. 2064–2072.
[27] Lucene. [n. d.]. Lucene Apache. https://lucene.apache.org/
[28] Xiao Luo, Chong Chen, Huasong Zhong, Hao Zhang, Minghua Deng, Jianqiang Huang, and Xiansheng Hua. 2020. A Survey on Deep Hashing Methods. *CoRR* abs/2003.03369 (2020).
[29] Natwar Modani, Rajeev Gupta, Guy M. Lohman, Tanveer Fathima Syeda-Mahmood, and Laurent Mignet. 2007. Automatically Identifying Known Software Problems. In *Proceedings of the 23rd International Conference on Data Engineering Workshops*. IEEE Computer Society, 433–441.
[30] Akira Moroo, Akiko Aizawa, and Takayuki Hamamoto. 2017. Reranking-based Crash Report Deduplication. In *The 29th International Conference on Software Engineering and Knowledge Engineering*, Xudong He (Ed.). KSI Research Inc. and Knowledge Systems Institute Graduate School, 507–510.
[31] Mozilla. 2012. Mozilla Crash Reporter. https://crash-stats.mozilla.org/
[32] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453.
[33] Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, and Alf Larsson. 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*. 240–250.
[34] Caitlin Sadowski and Greg Levin. 2007. Simhash: Hash-based similarity detection.
[35] Mark Sanderson. 2010. Test Collection Based Evaluation of Information Retrieval Systems. *Found. Trends Inf. Retr.* 4, 4 (2010), 247–375.
[36] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE)*. 118–121.
[37] Fran Silavong, Sean Moran, Antonios Georgiadis, Rohan Saphal, and Robert Otter. 2022. Senatus: a fast and accurate code-to-code recommendation engine. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 511–523.
[38] Roman Vasiliev, Dmitrij V. Koznov, George A. Chernishev, Aleksandr Khvorov, Dmitry V. Luciv, and Nikita Povarov. 2020. TraceSim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, FSE 2020*. 25–30.
[39] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for Similarity Search: A Survey. *CoRR* abs/1408.2927 (2014).
[40] Xiaofang Wang, Yi Shi, and Kris M. Kitani. 2016. Deep Supervised Hashing with Triplet Labels. In *Computer Vision - ACCV 2016 - 13th Asian Conference on Computer Vision*. 70–84.
[41] Ziming Zhang, Yuting Chen, and Venkatesh Saligrama. 2016. Efficient Training of Very Deep Neural Networks for Supervised Hashing. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 1487–1495.