BULLETIN of Cherkasy State Technological University

Journal homepage: https://bulletin-chstu.com.ua/en

Vol. 28 No. 4. 2023

UDC 004.421.22
DOI: 10.62660/2306-4412.4.2023.59-69

Deduplication of error reports in software malfunction: Algorithms for comparing call stacks

Serhii Pavlenko*

Postgraduate Student
Taras Shevchenko National University of Kyiv
01033, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0000-0003-4095-3925

Petro Kuliabko

PhD in Physical and Mathematical Sciences, Associate Professor
Taras Shevchenko National University of Kyiv
01033, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0000-0001-5411-6592

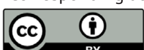
Abstract. In the software industry, the standard recognises automatic fault monitoring systems as mandatory for implementation. Considering the constant development of technologies and the high complexity of programmes, the importance of optimising processes for detecting and eliminating errors becomes a relevant task due to the need for reliability and stability of software. The purpose of this study is to conduct a detailed analysis of existing deduplication algorithms for reports from automatic systems collecting information about software failures. Among the algorithms considered were: the longest common subsequence method, Levenshtein distance, deep learning methods, Siamese neural networks, and hidden Markov models. The results obtained indicate a great potential for optimising processes of error detection and elimination in software. The developed comprehensive approach to the analysis and detection of duplicates in call stacks in failure reports allows for effectively addressing issues. The deep learning methods and hidden Markov models have demonstrated their effectiveness and feasibility for real-world applications. Effective methods for comparing key parameters of reports are identified, which contributes to the identification and grouping of recurring errors. The use of call stack comparison algorithms has proven critical for accurately identifying similar error cases in products with large audiences and high parallelism conditions. Siamese neural networks and the Scream Tracker 3 Module algorithm are used to determine the similarity of call stacks, including the application of recurrent neural networks (long short-term memory, bidirectional long short-term memory). Optimisation of report processing and clustering particularly enhances the speed and efficiency of responding to new failure cases, allowing developers to improve system stability and focus on high-priority issues. The study is useful for software developers, software development companies, system administrators, research groups, algorithm and tool development companies, cybersecurity professionals, and educational institutions

Keywords: automatic monitoring; fault detection systems; duplication removal; computer failures; analysis of interacting context structures

Article's History: Received: 31.08.2023; Revised: 16.11.2023; Accepted: 18.12.2023.

Suggested Citation:
Pavlenko, S., & Kuliabko, P. (2023). Deduplication of error reports in software malfunction: Algorithms for comparing call stacks. *Bulletin of Cherkasy State Technological University,* 28(4), 59-69. doi: 10.62660/2306-4412.4.2023.59-69.

*Corresponding author



Copyright © The Author(s). This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/)

INTRODUCTION

With the increasing complexity and volume of code in modern software systems, the probability of errors occurring during their operation is increasing. Even when using secure software development methods, errors can occur under inappropriate or unforeseen execution conditions, and reproducing them during testing is not always possible. This can have a negative impact on users and increase costs for further software maintenance. In particular, the growing importance of ensuring the security and stability of programmes indicates the importance of studying and optimising processes for detecting and fixing errors. However, with a large audience and complexity of programmes, there is a problem of effectively managing the flow of error reports coming from automatic monitoring systems. This creates the need for the development and implementation of effective methods and tools for analysing, filtering, and resolving these issues. Considering that modern software systems are used in various fields, from enterprises and e-commerce to medicine and finance, the stability and security of programmes are of great importance to end users and the business environment as a whole. Accurate resolution of this issue is a key aspect of ensuring the stability and efficiency of software development.

G.R. Sinha *et al.* (2021) and D. Feng (2022) investigated the issue of data deduplication. From the results of their study, it can be learnt that data deduplication is a data compression technology without loss that involves removing redundant data. This process involves removing duplicates, leaving only one physical copy that other copies can refer to. This leads to reduced storage costs and network bandwidth optimisation. Data deduplication methods are used to reduce the number of duplicates and repeated records. This technique improves the efficiency of data storage and usage and contributes to optimising data transmission over the network. Effective replacement of redundant fragments is achieved by identifying unique fragments and using the appropriate approach to removing duplicates.

V.S. Yakovyna and B.V. Uhrynovsky (2019) conducted an analysis of literary sources, which confirmed that the ageing of software leads to deterioration of performance and increased failures, negatively affecting its reliability and user satisfaction. The authors identified key characteristics of the phenomenon, such as effects, factors, metrics, and classification of ageing factors. The analysis of ageing modelling methods has shown that the development of hybrid approaches, combining analytical and measurement models, is a promising area for further research in this area. D. Medzatyi *et al.* (2023) emphasised that many software security approaches are aimed at avoiding complete failures, but the identification and classification of individual failures and vulnerabilities remains relevant. The literature review indicated that existing methods are not always suitable for classifying failures and vulnerabilities. Overall, this study aimed to develop a technology for identifying and classifying failures and vulnerabilities using surveys and rules based on response analysis to improve software security.

O.V. Shmatko and M.I. Myronenko (2018) discussed a testing technology that automatically sorts error reports in software. The uniqueness of this approach lies in the development of a software system for monitoring and detecting errors, which analyses error reports, automatically sorts them, and evaluates the reliability of the software. The paper by O.G. Trofymenko *et al.* (2023) emphasised that numerous risks accompany software development, and to maintain competitiveness, technology companies must effectively manage these risks. Software development projects are characterised by fast-paced and numerous changes, and risk classification helps to manage them effectively. Insufficient risk classification can lead to unforeseen problems and conflicts in risk management strategies, so adapted classifications that consider the specifics of software development and cybersecurity risks should be used.

The purpose of this study was to review existing software systems for optimisation tasks and determine their feasibility in the educational process, and to investigate systems that automatically collect data on software malfunction cases. The originality of the study lies in the examination of software for optimisation and justification of its application in educational processes. The feasibility of using technologies for detecting software vulnerabilities is generalised from the analysis of various studies, but this problem lacks a complete solution, namely due to the lack of generalised methods aimed at avoiding complete software ageing, deterioration of its performance, and increased failures, and insufficient risk management in the software development process.

MATERIALS AND METHODS

The study used the methods of longest common subsequence, Levenshtein, deep learning, Siamese neural networks and hidden Markov models. The longest common subsequence method was one of the first to be used to determine a similarity between call stack fragments (Castelluccio *et al.*, 2017). It allowed identifying sequences of functions present in both call stacks, regardless of their order. These algorithms can be used to determine the degree of similarity of programme call stacks, which is an important criterion for establishing relationships between programmes, detecting anomalies, and in other areas of software analysis.

The Levenshtein method was the next step as an improvement over the longest common subsequence method. Its logical use is due to the fact that the call stack is an ordered set of frames, and is represented as a sequence of insertion, replacement, and deletion operations. The Levenshtein algorithm is modified to account for the peculiarities of call stacks, in particular, the costs of operations are statistically determined based on the training data set.

Deep learning has shown potential for use in determining the similarity of call stacks in crash reports (Ebrahimi *et al.*, 2016). It effectively considers the context of the frame in the call stack, which allows for a more accurate determination of similarity between them. One of the prospects of this approach is the ability to automatically extract key features, facilitating efficient processing and measurement of similarity. In addition, deep learning algorithms have shown robustness to noise and data variability, providing accurate measurements of similarity in diverse and unpredictable data sets.

Siamese neural networks have proven to be an effective approach to determining the similarity of call stacks in crash reports using deep learning (Dang *et al.*, 2012). Using a Siamese architecture, the method evaluated the similarity of stacks, considering their internal structure. Key elements include converting the call stack into a vector form, using a recurrent neural network to obtain a transformed vector representation, and using a classifier to assess the similarity between stacks. With the Siamese architecture, this method provided an effective assessment of call stack similarity, with the major advantages being a smaller training sample size and the ability to automatically extract key features for more accurate comparison.

The hidden Markov model method was an approach to detecting duplicates in crash reports (van Tonder *et al.*, 2018). This algorithm was based on the idea of creating a separate hidden Markov model for each cluster, allowing for efficient determination of a specific stack belonging to a particular cluster. Each stack was considered as a discrete sequence of states or function calls, and the determination of the number of observed and hidden states was done using unique frames of the stacks in the cluster.

The general matrix H for the entire set of call stacks is calculated using the formula (Castelluccio *et al.*, 2017) (1):

$$H_{i,j} = max\begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - d \\ H_{i,j-1} - d \end{cases}, \quad (1)$$

where $H(i, j)$ – optimal value of the algorithm for the i and j elements of the respective subsequences; $S(i, j)$ – value of comparing these elements; d – fixed penalty for a missing or extra function.

To calculate the result of the comparison, it is necessary to consider how important a given function $P(f)$ is (Castelluccio *et al.*, 2017) (2):

$$P_1(f) = 1 - \frac{Number\ of\ stacks\ containing\ f}{Total\ number\ of\ stacks}. \quad (2)$$

It is also necessary to consider the position of the sought function a_i in the stack, relative to the top, i.e., $P(a_i)$. The closer it is to the top, the higher the weight (Castelluccio *et al.*, 2017) (3):

$$P_2(a_i) = 1 - \frac{i}{Total\ number\ of\ frames}, \quad (3)$$

where i – interval between a_i – i-th frame of the stack being compared to b_j – j-th frame of the stack, which is introduced in the next formula (Castelluccio *et al.*, 2017) (4):

$$P_3(a_i, b_j) = e^{-|i-j|}. \quad (4)$$

The comparison function looks as follows (Castelluccio *et al.*, 2017) (5):

$$S_{i,j} = max\begin{cases} P_1(f) * P_2(a_i) * P_3(a_i, b_j), if\ a_i = b_j = f \\ 0, otherwise \end{cases}. \quad (5)$$

The weight calculation of the function has the following form (Rosenberg & Moonen, 2018) (6, 7):

$$\omega(f_i) = l\omega_{\alpha}(f_i) * g\omega_{\beta\gamma}(f_i), \quad (6)$$

where $\omega(f_i)$ – weight of function f_i; $g\omega_{\beta\gamma}(f_i)$ – global weight of function f_i among all stacks accumulated in the base; α, β, γ – coefficients for adjusting the algorithm.

$$l\omega_{\alpha}(f_i) = \frac{1}{i^{\alpha}}, \quad (7)$$

where i – frame number.

The calculation of the total weight is done using the $TF - IDF$ ranking function (Rosenberg & Moonen, 2018) (8):

$$g\omega_{\beta\gamma}(f_i) = \frac{1}{1+e^{-x}} * (\beta * (IDF(f_i) - \gamma), \quad (8)$$

where $TF(f_i) = 1$, $IDF(f_i)$ – inverse document frequency; β, γ – parameters for tuning $IDF(f_i)$.

The calculation of the feature vector based on the two obtained vectors is (Dang *et al.*, 2012) (9):

$$features(v_1, v_2) = (|v_1 - v_2|, \frac{(v_1+v_2)}{2}, v_1 \times v_2), \quad (9)$$

where $v_1 = biLSTM(C1)$, $v_2 = biLSTM(C2)$, despite the fact that $C1, C2$ are call stacks.

The stack as a discrete sequence of states or function calls (van Tonder *et al.*, 2018) (10):

$$S = \{f_1, f_2, \ldots, f_m\}. \quad (10)$$

The structure of the hidden Markov model is determined as follows (van Tonder *et al.*, 2018) (11):

$$\underline{V} = \{v_1, \ldots, v_M\}, \quad (11)$$

where M – number of observable states.

The set of observations is the set of hidden states of the model (van Tonder *et al.*, 2018) (12):

$$S = \{s_1, \ldots, s_N\}, \quad (12)$$

where N – number of hidden states.

RESULTS

Collection and analysis of error reports in software. Modern information systems are rapidly increasing in code volume and complexity, and most coding errors and unforeseen conditions are not detected during the testing phase. After software release, its further maintenance requires significant resources to adapt to new operating systems, fix critical errors, and expand functionality. User reports are not efficient enough due to objective limitations. Thus, automatic error report collection is a necessary part of software development, provided by systems that monitor programmes for failures and send reports to the server when they occur (Brodie *et al.*, 2005). These reports include memory dumps, log files, configuration files, and other characteristics of software operation, allowing for quicker and more effective problem resolution, collecting data about the software usage environment, measuring its stability, and prioritising failures. The intensity of the flow of error reports depends on the number of software users and can reach millions per day. The task of deduplicating reports arises from the large number of duplicates and outdated reports. Accurate deduplication is essential for correct statistics and prioritisation of failures and allowing engineers to use their time more efficiently by eliminating the need to review recurring failures and control defect fixes.

Overall, a failure is defined as the inability of software or its component to perform its functions or meet defined criteria, such as speed or resource usage. Meanwhile, a thread's call stack represents a data structure that contains information about the sequence of function calls made by a single execution thread in a programme. The thread's call stack stores information about the current state of each function in that thread, including the location of the programme counter and the values of local variables (Bartz *et al.*, 2008). It can also contain information about the return point within the thread, allowing the programme to correctly resume after the function completes. In a programme with parallel execution, there may be multiple thread call stacks, each operating independently, but in case of an error or failure, tracing the call stack of that thread indicates the sequence of functions that caused the problem. In addition, there is the concept of a programme memory dump, which is a file that captures the state of the programme's memory at a specific moment in time (Wrembel, 2022). It can include the entire memory area the programme uses at that particular moment.

When it comes to existing algorithms for clustering crash reports, they can be divided into two categories depending on the information they use. In the first group, algorithms use metadata about crashes, while in the second group, they use the call stack of the thread where the crash occurred. For a better understanding of the crash report clustering processes, it is worth using the schematic diagram presented in Figure 1.

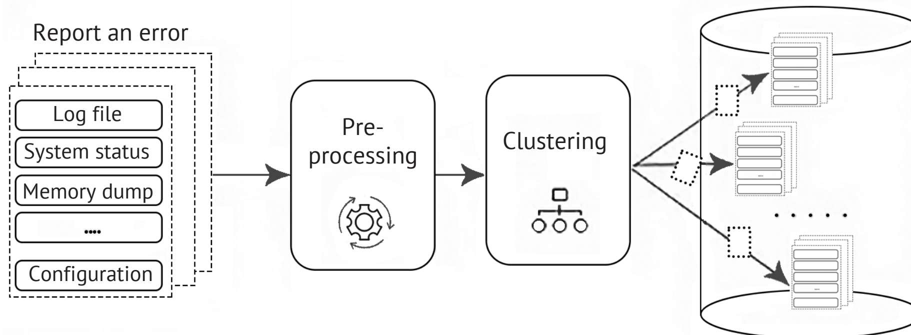


Figure 1. Schematic representation of the crash report clustering process
Source: compiled by the authors

Both approaches have their advantages and disadvantages. Among the disadvantages, the need to collect memory dumps can be highlighted, which takes time and increases the size of the report. Furthermore, the large sizes of dumps and symbol files, their complex restoration process, and possible impact on the computational power of the user's computer are disadvantages. On the other hand, the call stack provides comprehensive information about the cause of the crash and allows for additional investigation of the programme's state. In any case, both methods can be used separately or in combination. A typical example of a call stack for Linux systems can also be considered (Table 1). Frames 1 to 9 describe the programme's logic, while frames 10 and 11 represent system calls for thread initialisation.

Table 1. Example of a crash call stack in Mozilla

Frame	Module	Function signature	Path to the source file
1	libxul.so	mz::lrs::NativeCA::NotifySurfaceReady()	RenderComposNative.cpp:291
2	libxul.so	mz::lrs::NativeOGL::Unbind()	RenderComposNative.cpp:558

Continued Table 1

Frame	Module	Function signature	Path to the source file
3	webrender.so	webrender::Renderer::draw_frame	renderer/mod.rs:4412
4	libxul.so	nsThread::ProcessNextEvent(bool, bool*)	threads/nsThread.cpp:1233
5	libxul.so	MessageLoop::RunInternal()	base/message_loop.cc:381
6	libxul.so	MessageLoop::RunHandler()	base/message_loop.cc:374
7	libxul.so	nsThread::ThreadFunc(void*)	threads/nsThread.cpp:391
8	libxul.so	std::sys::unix::Thread::thread_start	sys/unix/thread.rs:108
9	firefox-bin	set_alt_start(PthreadCreateParams*)	pthread_interposer.cpp:80
10	libc.so.6	start_thread	
11	libc.so.6	clone3	

Notes: Frame 10 and frame 11 are standard functions, and their path to the source file is not important for this study.
Source: compiled by the authors

Before any actions with the call stack, a very important step is preprocessing, the main task of which is to remove insignificant data from the stack. Preprocessing may include actions such as partially removing frames of recursive calls, identifying runtime and system modules, removing fault handling frames, marking frames according to module versions, identifying insignificant or "reliable" functions, and formulating a "dictionary" of functions in the call stack. Another important criterion is that the function where the crash occurred should be identical. Nevertheless, identifying this function is not always straightforward and may not be at the top of the stack due to error handling, platform or runtime code, or changes in the product's version. In any case, the further away a function is from the top of the stack, the less likely it is to be relevant to the crash, and the more times it may appear in the reports. For example, thread initialisation functions (Fig. 2).

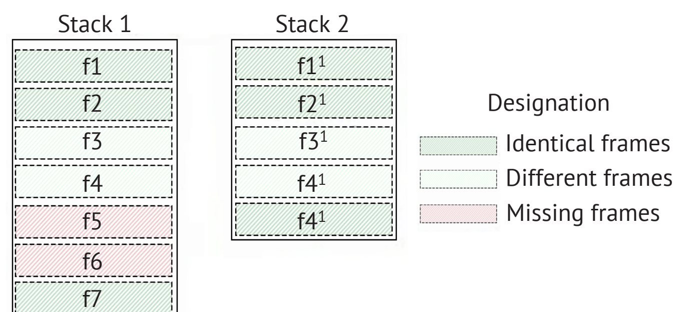


Figure 2. Example of two call stacks that are compared
Source: compiled by the authors

In addition to the function name, another important aspect is parameters such as translation unit, implementation location, and data types, which also affect comparison and evaluation. There may be multiple stacks that lead to a crash in the same function. Detecting such groups of stacks is important for comprehensive information and effective fault removal.

Algorithms for processing text information and measuring similarity. Solving the task of comparing call stacks is a relatively new topic that uses various adapted algorithms. Initial attempts were based on simple string comparison algorithms, from straightforward sequence comparison to using regular expressions. Subsequently, ranking-based algorithms were introduced, which proved to be more flexible and accurate. In addition, attempts have been made to use graph similarity algorithms. Overall, in recent years, there has been a significant variety of algorithms, including those based on machine learning methods.

There is an algorithm that uses a modified longest common subsequence (LCS) method. However, its application also requires using a variation of the Needleman-Wunsch algorithm, which allows any possible gaps but does not allow frame replacements. Considering that the function where the failure occurred is most likely to be at the top of the stack, it is recommended to introduce a parameter that determines the position relative to the top. This algorithm uses dynamic programming and calculates a common matrix for the entire set of call stacks (Formula 1).

The next stage in improving the previous method is the ReBucket algorithm. Its main principle is also based on finding LCS, but it differs from the previous one in the approach to the comparison function Si, j and the

absence of fixed penalties *d*. Its modification is known as the position-dependant model (PDM). The key idea is that functions located far from the top of the stack should have less influence on the comparison result. The algorithm allows adjusting the weight of the influence using respective parameters. The concept of "Alignment Offset" is also introduced to account for missing or extra functions.

To express the difference between two call stacks as a number, an adapted Levenshtein distance algorithm can also be used. Its use for this task is logical, as the call stack is an ordered set of frames. The basis is the classic Levenshtein distance calculation algorithm, which uses only modifications of insertion, deletion, and replacement. For the given stacks, transposition is not considered since the order of frames is important. The result of the Levenshtein distance calculation will be the minimum number of modifications. The maximum length of the two stacks will be used to normalise the obtained value so that the result is within 0 to 1.

Two essential modifications to the algorithm should be mentioned. First, unlike the classical version of the Levenshtein algorithm, where the cost of deletion, replacement, and insertion operations is considered the same, this cost should be determined separately for each operation using a statistical model built on the training data. Second, in the case of text processing, there is a constant set of characters, while when working with call stacks, a frame is an analogue of a character. The frame includes the module, address, and function signature. These parameters are also considered when determining operations with corresponding costs. An example of a frame could look like this: "Module: example_module; Function address: 0x12345678; Function signature: void example_function(int a, char b)".

Methods of deep learning are widely used for analysing various data, which makes them potentially effective for determining the similarity of call stacks in crash reports. Although these methods are actively used for crash report deduplication tasks, their application for measuring the similarity of call stacks is less common. It is worth noting the promising prospects of such an approach. For example, the ability to consider the frame context. Deep learning can effectively evaluate the relationships between different call stack frames, which allows for more accurate determination of similarity. An important criterion is the automatic extraction of key features. Deep learning models can automatically determine important features from data, providing more efficient processing and consideration of similarity. In addition, there is robustness to noise and data variability. Deep learning algorithms demonstrate greater robustness to noise and data variability, allowing for precise similarity measurements even in complex or unpredictable data sets (Yang *et al*., 2023). It is worth using feature embedding, considering the possibility of deriving semantic similarity between call stack frames. Flexibility is also an important aspect, as deep learning algorithms can be adapted to different types of data, such as text, images, and sounds, making them versatile for various applications.

Siamese neural networks, the S3M method, and hidden Markov models. The Scream Tracker 3 Module (S3M) can be considered as the first effective example of using deep learning for calculating the similarity of call stacks. In general, the Siamese architecture consists of at least two identical convolutional neural networks and an evaluation module. Here, identity means that the networks have the same configuration and weight coefficients. Convolutional networks receive call stacks as input and return feature vectors, which are passed to the evaluation module, which returns a number characterising their similarity in the range from 0 to 1.

Classical neural networks are also used to solve clustering tasks, but for this, predefined clusters and a significant training sample are required. In the case of Siamese neural networks, their task is to estimate the similarity of data from a specific domain, so they require a smaller training sample that should consider the diversity of data with which the network will work (Esteves *et al*., 2023). As for the S3M algorithm itself, its first step is to convert the call stack into a vector form. To do this, functions in frames are cropped and tokenised. Cropping is used to minimise the size of the token vocabulary, for example, module or class names can be truncated. Minimising the vocabulary helps improve comparison. Furthermore, having a vector representation of stacks, it is worth using recurrent neural networks (RNN) to obtain a transformed vector representation. Since this process requires substantial resources, the obtained representation is stored in the system, and for further comparisons, it is necessary to compute only for the input call stack.

The next step is to use a classifier for the pair of transformed call stack representations. Since these stacks can be quite large, the algorithm uses an architecture called long short-term memory (LSTM) in the RNN. This network iteratively works with data (tokens), changing its internal state at each iteration and generating an output vector-result. This allows the RNN to "remember" and consider the previous context, processing each subsequent token. Generally, RNN and its LSTM architecture were initially developed for working with text but later found application in other areas.

The classical LSTM architecture has a drawback in that the network considers only tokens that are located before the token with which it is currently working at a specific time (Islam *et al*., 2021). However, in the case of call stacks, frames $f_{k+1...n}$ located after the current frame in the stack also have a significant impact. The situation arises as follows: the direct order of frames is important for the overall assessment of stack similarity, and the reverse order is important for assessing the impact of a specific frame on the entire call stack. To consider this feature, a modification of LSTM called

bidirectional LSTM (BiLSTM) should be used. In general, this is simply two classical LSTMs, where one processes the data in the forward direction, and the other in the reverse. The result is obtained by concatenating the two vector-results of these two LSTM networks.

The next step is to calculate a feature vector based on the two obtained vectors (9), which should consider the common and different aspects of the two vectors. The final step is that the obtained result is processed by a two-layer neural network with a rectified linear unit activation function. Considering the system's ability to store computed feature vectors for already processed stacks, the amortised time complexity of the algorithm is equal to $O(dh)$, where d and h represent the size of the input and hidden layers of the two-layer neural network that performs the final similarity estimation. A drawback of this approach, like other deep learning methods, is the need for a significant training sample.

Another interesting method is the hidden Markov model (HMM). Typically, HMM is widely used for speech recognition, DNA sequence analysis, and text processing, demonstrating impressive results in learning data sequences. An example of HMM is shown below (Fig. 3).

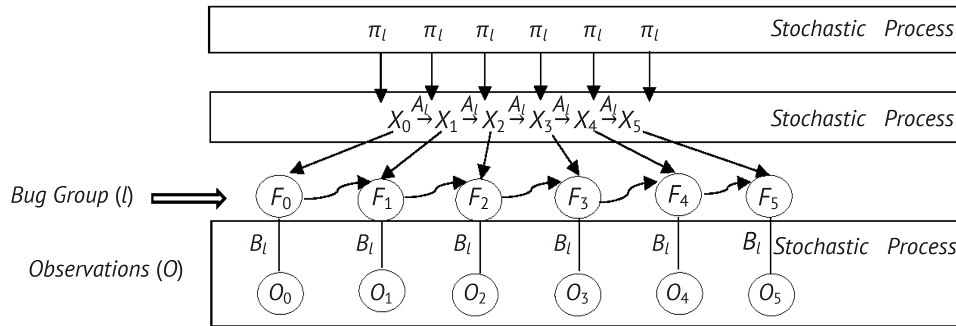


Figure 3. Detecting duplicate crash reports using HMM

Source: N. Ebrahimi *et al.* (2016)

This algorithm is based on the idea of creating a separate hidden Markov model for each cluster, which allows determining which cluster the given stack belongs to. The stack can be represented as a discrete sequence of states or function calls (10). To determine the number of observable states (11), all unique frames of stacks for all reports in the cluster should be found. The number of hidden states (12) is determined by the method of separate selection for each case within $N = [5, 10, 15, \ldots, 50]$. Other elements should also be considered. For example, A is a matrix of transition probabilities for a set of states S of size $N \times N$. Besides, B is a matrix of observation probability distribution of size $N \times M$. A value $\pi = \{\pi i\}$ is the initial distribution. The HMM model is denoted as $\lambda = (A, B, \pi)$, where A represents the probability distribution of states and transitions of the system in a Markov process. Training HMM based on discrete observations $O(O_0, O_2, \ldots, O_{T-1})$ is the maximisation of the probability $P(O|\lambda)$ among the parameters A,B,π.. The mentioned model is built for all clusters. One important feature of this algorithm is the need for historical data. It is interesting to note that this algorithm is an improvement of CrashAutomata, built on the generation of finite automata.

DISCUSSION

It is worth considering other studies in this area for a more detailed analysis of detecting duplicate error reports in software and algorithms for comparing call stacks. For example, R. Van Tonder *et al.* (2018) considered automated dynamic testing tools, including fuzzers that generate a large number of erroneous or boundary input data to provoke crashes. To reduce the number of duplicate crash reports, they use various heuristics, such as stack hashes, etc. However, after removing duplicates, there remains a substantial number of unique crashes that may correspond to the same error (have a common cause). The paper proposes a method of semantic clustering of crashes, which uses programme transformations to accurately group crashes. Instead of fast but inaccurate heuristics, the method uses approximate bug fixes through minor modifications of the programme's source code based on patch patterns and semantic feedback. This approach is suitable for general classes of bugs and outperforms built-in deduplication methods for next-generation fuzzers. Like in this study, the aforementioned study addresses the issue of deduplicating error reports. Nevertheless, this study concerns deduplicating crash reports received from users during normal usage, while in the other case, it deals with general reports of system malfunctions and their detection during testing stages. Furthermore, in the former study, unlike this one, semantic clustering of crashes is used. In this study, crash grouping is based on the similarity of call stacks.

C.M. Rosenberg and L. Moonen (2018) focused on the problem of grouping logs of software system operations related to identical faults. They investigated whether such an approach could be applied to logs of a system operating in real conditions. They also evaluated the impact of dimensionality reduction and cluster merging criteria on the quality of automatic clustering.

The authors used a method that replicates and extends the work on clustering of system logs. They considered the inclusion of dimensionality reduction methods and three cluster merging criteria. The results showed that fault detection in continuous deployment logs through clustering is reasonable, and fault detection through automatic clustering improves when dimensionality reduction is applied, which increases its robustness to different input data. The common aspects are the consideration of clustering methods. However, the methods used to achieve the purpose differ in the studies and they use different data to reach it.

Furthermore, S. Gupta and S. Gupta (2021) emphasised that defects are an integral part of software projects. In open-source projects, bug reports are stored in open bug repositories. When a new report arrives, a person called a "triager" analyses it before assigning it to the responsible developer, trying to determine if it is not a duplicate. The problem of duplicates complicates the software maintenance process, and the paper investigates the issue of detecting duplicate defect reports. It also classifies and analyses existing research on duplicate defect report detection, noting the advantages, limitations, and defining key areas for future investigation in this area. It can be concluded that both studies analyse the problem of deduplicating crash and software defect reports. However, in this study, algorithms are considered that allow automating the clustering of reports based on call stacks, rather than descriptions, unlike the other study. In addition, this study not only focuses on detecting duplicate error reports but also proposes various ways to address this issue.

S. Mukhtar *et al.* (2023) argued that defect reports can vary in granularity: some are exhaustive, while others are quite brief. In such cases, duplicate defect reports can be a useful resource for enriching defect descriptions. Nevertheless, existing methods for summarising descriptions mainly focus on individual defect reports. In this paper, the authors explored the summarisation of duplicate software defect reports to obtain an informative summary, reducing redundant phrases. They applied various text summarisation methods to duplicate defect reports, comparing their effectiveness and determining the best method. The experimental results confirm that the extractive multi-document method is the most effective, providing detailed summaries of duplicate defect reports. This helps advance defect report summarisation techniques based on unique information from duplicates. Thus, both studies address the issue of duplicate crash and software defect reports. However, this study examines clustering methods based on the similarity of call stacks and failure metadata obtained automatically, while the other study focuses on a multi-document method based on defect reports. The idea of complementing information based on duplicates is also interesting.

S. Jahan and M.M. Rahman (2022) analysed a large number of defect reports from three open-source systems, identified textually similar and distinct duplicates, and evaluated the effectiveness of existing duplicate detection methods. The authors discovered that existing methods perform poorly for textually distinct duplicates, as they often miss important components, leading to low efficiency. They also used domain-specific embeddings to improve the efficiency of traditional duplicate detection methods. The results obtained showed that such an approach of combining traditional methods with modern large language models achieves new record results, highlighting the potential of such a combination to improve software engineering tasks. The commonality in both studies lies in the use of methods for detecting duplicate error reports. Nonetheless, the difference lies in the approaches to detecting these duplicates and the data used from the reports for this purpose.

S. Qian *et al.* (2023) noted that a large number of new software defects constantly arise, creating a workload for those who fix them. Therefore, effective defect deduplication is of great importance in software development. This study begins with an examination of the available literature, including an analysis of research trends, mathematical models, methods, and widely used datasets. Further, the authors generalised the overall process of using methods, analysing them in terms of information obtained during execution and information taken from error reports. They also provide a detailed review of the methods used in relevant works. Ultimately, a detailed comparison of practical findings obtained in different studies is provided, based on indicators such as usage methods, datasets, accuracy, and completeness, among others. The main common aspect between the two studies is that they both address deduplication of errors in software. Both studies analyse different mathematical models and methods. However, the metrics, results, and the actual methods of detecting duplicates in the studies differ.

T. Zhang *et al.* (2023) focused on duplicate bug report detection (DBRD), which is a problem in science. The researchers proposed various methods for more accurate duplicate detection using both conventional and deep approaches. The study showed that deep models are not always more effective, but conventional methods are limited in that they do not capture the semantics of error reports. To overcome these limitations, they proposed using a large language model to enhance the conventional DBRD approach. This approach, called Cupid, combines the best conventional method with the ChatGPT (Generative Pre-trained Transformer) large language model, which was used to extract important information from error reports. In experiments, Cupid showed new best results, indicating the potential of combining large language models to improve performance in software engineering tasks. The common

aspects include obtaining methods for detecting duplicate error reports. Nevertheless, unlike this study, the other study uses large language models (specifically ChatGPT) and has a specific approach to detecting duplicate reports. In addition, the authors' approach is based on defect reports, whereas this study uses automatically generated crash reports.

Thus, like in this study, the above-mentioned studies address the deduplication of crash and software defect reports. They also share certain methods for detecting such duplicates and clustering. However, in each study, the detection of duplicate reports and recommendations for their resolution have their own features and differences.

CONCLUSIONS

Research on automatic detection and correction of errors in software has proven to be important in the context of the modern industry. This study examined the achievement of this goal using methods such as Longest Common Subsequence, Levenshtein distance, deep learning, Siamese neural networks, and hidden Markov models. Each of the described algorithms has its advantages and disadvantages. LCS-based algorithms determine the longest sequence of method calls present in both call stacks, regardless of their order, and are characterised by their simplicity and efficiency. Another algorithm used in many approaches for measuring the similarity of call stacks is Levenshtein distance. It calculates the minimum number of insertions, deletions, and substitutions required to transform one call stack into another. Hidden Markov models are probabilistic models that can be used to represent the sequence of method calls in a call stack. Siamese neural networks have been applied to measure the similarity of call stacks and have shown promising results. Using deep learning approaches can help identify complex dependencies between call stacks and provide more accurate clustering results.

Overall, the results indicate a substantial increase in interest in this issue, and the use of recent advances in deep learning opens up promising ways to detect and deduplicate errors. The developed comprehensive approach to the analysis and detection of duplicate call stacks considers the advantages and limitations of different methods, providing the opportunity to choose the optimal approach for a specific task. Considering the conclusions drawn during the study, several practical recommendations can be made for detecting and correcting errors in software, and for further development in this field. The methods and algorithms discussed highlight the importance of using a comprehensive approach to error analysis. Given the effectiveness of deep learning methods in measuring the similarity of call stacks, further improvement of these methods and their consideration in software lifecycle management is recommended.

It is recommended to develop universal algorithms that accurately identify similar error cases in runtime conditions on different operating systems and contribute to further optimisation of crash report processing. In addition, exploring the possibilities of using hybrid clustering models that combine stack-based approaches and metadata is recommended. This can help accommodate changes in implementation between different versions of the software. Deep learning methods are promising in terms of their ability to learn during use, which will allow the system to adapt to a specific product. Among the further research areas, the following can be highlighted: development of methods to ensure confidentiality and integration of deduplication into security systems; examination of algorithms that consider the specifics of different types of programmes and their calls; creation of effective filters to identify critical errors and filter out irrelevant or duplicate reports, etc.

ACKNOWLEDGEMENTS

None.

CONFLICT OF INTEREST

None.

REFERENCES

[1] Bartz, K., Stokes, J.W., Platt, J.C., 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 (Sysml'08)*. Berkeley: USENIX Association. doi: 10.5555/1855895.1855896.

[2] Brodie, M., Ma, S., Lohman, G., Mignet, L., 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). Seattle: Institute of Electrical and Electronics Engineers. doi: 10.1109/ICAC.2005.49.

[3] Castelluccio, M., Sansone, C., Verdoliva, L., & Poggi, G. (2017). Automatically analyzing groups of crashes for finding correlations. In *ESEC/FSE 2017: Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 717-726). New York: Association for Computing Machinery. doi: 10.1145/3106237.3106306.

[4] 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 *2012 34th international conference on software engineering (ICSE)* (pp. 1084-1093). Zurich: Institute of Electrical and Electronics Engineers. doi: 10.1109/ICSE.2012.6227111.

[5] Ebrahimi, N., Islam, S., Hamou-Lhadj, A., & Hamdaqa, M. (2016). An effective method for detecting duplicate crash reports using crash traces and hidden Markov models. In *CASCON '16: Proceedings of the 26th annual international conference on computer science and software engineering* (pp. 75-84). Riverton: IBM Corp. doi: 10.5555/3049877.3049885.

[6] Esteves, J., Costa, R., Zhou, Y., & Almeida, A. (2023). An exploratory analysis of methods for real-time data deduplication in streaming processes. In *DEBS '23: Proceedings of the 17th ACM international conference on distributed and event-based systems* (pp. 91-102). New York: Association for Computing Machinery. doi: 10.1145/3583678.3596898.

[7] Feng, D. (2022). *Data deduplication for high performance storage system*. Singapore: Springer. doi: 10.1007/978-981-19-0112-6_2.

[8] Gupta, S., & Gupta, S. (2021). A systematic study of duplicate bug report detection. *International Journal of Advanced Computer Science and Applications*, 12(1), 578-589. doi: 10.14569/IJACSA.2021.0120167.

[9] Islam, S., Hamou-Lhadj, A., Koochekian Sabor, K., Hamdaqa, M., & Cai, H. (2021). EnHMM: On the use of ensemble HMMs and stack traces to predict the reassignment of bug report fields. In *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)* (pp. 411-421). Honolulu: Institute of Electrical and Electronics Engineers. doi: 10.1109/SANER50967.2021.00045.

[10] Jahan, S., & Rahman, M.M. (2022). Towards understanding the impacts of textual dissimilarity on duplicate bug report detection. In *2023 IEEE international conference on software analysis, evolution and reengineering (SANER)* (pp. 25-36). Taipa: Institute of Electrical and Electronics Engineers. doi: 10.1109/SANER56733.2023.00013.

[11] Medzatyi, D., Voichur, Yu., & Voichur, O. (2023). Technology of identification and classification of software failures and vulnerabilities. *Measuring and Computing Devices in Technological Processes*, 1, 53-57. doi: 10.31891/2219-9365-2023-73-1-8.

[12] Mukhtar, S., Primadani, C.C., Lee, S., & Jung, P. (2023). A comparison of summarization methods for duplicate software bug reports. *Electronics*, 12(16), article number 3456. doi: 10.3390/electronics12163456.

[13] Qian, C., Zhang, M., Nie, Y., Lu, S., & Cao, H. (2023). A survey on bug deduplication and triage methods from multiple points of view. *Applied Sciences*, 13(15), article number 8788. doi: 10.3390/app13158788.

[14] Rosenberg, C.M., & Moonen, L. (2018). Improving problem identification via automated log clustering using dimensionality reduction. In *ESEM '18: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement* (pp. 1-10). New York: Association for Computing Machinery. doi: 10.1145/3239235.3239248.

[15] Shmatko, O.V., & Myronenko, M.I. (2018). Information technology of depending of errors software. *Scientific Works of Kharkiv National Air Force University*, 2(56), 120-125. doi: 10.30748/zhups.2018.56.17.

[16] Sinha, G.R., Thwel, T.Th., Mohdiwale, S., & Shrivastava, D.P. (2021). Introduction to data deduplication approaches. In T.Th. Thwel & G.R. Sinha (Eds.), *Data deduplication approaches. Concepts, strategies, and challenges* (pp. 1-15). Cambridge, Massachusetts: Academic Press. doi: 10.1016/C2020-0-00104-0.

[17] Trofymenko, O.G., Loginova, N.I., Teslenko, P.O., Savielieva, O.S., & Poliakov, V.M. (2023). Classification of software project risks. *Visnyk of Kherson National Technical University*, 3(86), 119-128. doi: 10.35546/kntu2078-4481.2023.3.15.

[18] van Tonder, R., Kotheimer, J., & Le Goues, C. (2018). Semantic crash bucketing. In *ASE '18: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering* (pp. 612-622). New York: Association for Computing Machinery. doi: 10.1145/3238147.3238200.

[19] Wrembel, R. (2022). Data integration, cleaning, and deduplication: Research versus industrial projects. In E. Pardede, P.D. Haghighi, I. Khalil & G. Kotsis (Eds.), *Proceedings of the 24th international conference "Information integration and web intelligence"* (pp. 3-17). Cham: Springer. doi: 10.1007/978-3-031-21047-1_1.

[20] Yakovyna, V.S., & Uhrynovskyi, B.V. (2019). Software aging in the context of its reliability: A systematic review. *Scientific Bulletin of UNFU*, 29(5), 123-128. doi: 10.15421/40290525.

[21] Yang, C., Chen, J., Fan, X., Jiang, J., & Sun, J. (2023). Silent compiler bug de-duplication via three-dimensional analysis. In *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis* (pp. 677-689). New York: Association for Computing Machinery. doi: 10.1145/3597926.3598087.

[22] Zhang, T., Irsan, I.C., Thung, F., & Lo, D. (2023). Cupid: Leveraging ChatGPT for more accurate duplicate bug report detection. *Cornell University*, 37(4), article number 1. doi: 10.48550/arXiv.2308.10022.

Дедублікація звітів про помилки в роботі програмного забезпечення: алгоритми порівняння стеків викликів

Сергій Васильович Павленко
Аспірант
Київський національний університет імені Тараса Шевченка
01033, вул. Володимирська, 60, м. Київ, Україна
https://orcid.org/0000-0003-4095-3925

Петро Петрович Кулябко
Кандидат фізико-математичних наук, доцент
Київський національний університет імені Тараса Шевченка
01033, вул. Володимирська, 60, м. Київ, Україна
https://orcid.org/0000-0001-5411-6592

Анотація. В індустрії системи автоматичного моніторингу збоїв у програмному забезпеченні визнані обов'язковим для впровадження стандартом. Враховуючи постійний розвиток технологій і високу складність програм, важливість оптимізації процесів виявлення та усунення помилок стає актуальним завданням завдяки потребі у надійності та стабільності програмного забезпечення. Мета даного дослідження полягає в детальному аналізі існуючих алгоритмів дедублікації звітів систем автоматичного збору інформації про збої у роботі програмного забезпечення. Серед розглянутих алгоритмів, були наступні: метод найдовшої спільної підпослідовності, відстань Левенштейна, методи глибинного навчання, сіамські нейронні мережі та метод прихованих марковських моделей. Отримані результати свідчать про великий потенціал оптимізації процесів виявлення та усунення помилок в програмному забезпеченні. Розроблений комплексний підхід до аналізу та виявлення дублікатів стеків викликів у звітах про збої дозволяє ефективно вирішувати проблеми. Використані методи глибинного навчання та прихованих марковських моделей проявили свою ефективність та можливість використання в реальних умовах. Зазначено ефективні способи порівняння ключових параметрів звітів, що сприяє ідентифікації та групуванню повторюваних помилок. Використання алгоритмів порівняння стеків викликів виявилося критичним для точного виявлення схожих випадків помилок у продуктах з великою аудиторією та умовами високої паралельності. Сіамські нейронні мережі та алгоритм Scream Tracker 3 Module використовуються для визначення подібності стеків викликів, зокрема, застосовуються рекурентні нейронні мережі (long short-term memory, bidirectional long short-term memory). Оптимізація обробки та кластеризації звітів значно підвищує швидкість та ефективність реагування на нові випадки збоїв, дозволяючи розробникам удосконалити стабільність системи та зосередитися на проблемах високого пріоритету. Дослідження корисне для розробників програмного забезпечення, компаній з розробки ПЗ, системних адміністраторів, дослідницьких груп, компаній з розробки алгоритмів та інструментів, фахівців у галузі кібербезпеки, а також освітніх установ

Ключові слова: автоматичний моніторинг; системи виявлення недоліків; усунення повторів; комп'ютерні збої; аналіз структури взаємодіючих контекстів