An HMM-Based Approach for Automatic Detection and Classification of Duplicate Bug Reports

Neda Ebrahimi*, Abdelaziz Trabelsi, Md. Shariful Islam, Abdelwahab Hamou-Lhadj and Kobra Khanmohammadi

Department of Electrical and Computer Engineering, Concordia University, Montréal, QC, Canada

Abstract

Context: Software projects rely on their issue tracking systems to guide maintenance activities of software developers. Bug reports submitted to the issue tracking systems carry crucial information about the nature of the crash (such as texts from users or developers and execution information about the running functions before the occurrence of a crash). Typically, big software projects receive thousands of reports every day.

Objective: The aim is to reduce the time and effort required to fix bugs while improving software quality overall. Previous studies have shown that a large amount of bug reports are duplicates of previously reported ones. For example, as many as 30% of all reports in for Firefox are duplicates.

Method: While there exist a wide variety of approaches to automatically detect duplicate bug reports by natural language processing, only a few approaches have considered execution information (the so-called stack traces) inside bug reports. In this paper, we propose a novel approach that automatically detects duplicate bug reports using stack traces and Hidden Markov Models.

Results: When applying our approach to Firefox and GNOME datasets, we show that, for Firefox, the average recall for Rank k =1 is 59%, for Rank k=2 is 75.55%. We start reaching the 90% recall from k=10. The Mean Average Precision (MAP) value is up to 76.5%. For GNOME, The recall at k=1 is around 63%, while this value increases by about 10% for k=2. The recall increases to 97% for k=11. A MAP value of up to 73% is achieved.

Conclusion: We show that HMM and stack traces are a powerful combination for detecting and classifying duplicate bug reports in large bug repositories.

Keywords: Duplicate Bug Reports; Stack Traces; Hidden Markov Models; Machine Learning, Mining Software Repositories.

1. Introduction

A Bug Tracking System (BTS) is used by development and management teams to keep track of software bugs and resolutions. There are many bug tracking systems, both commercial and open-source. Among the freely available open-source systems, Bugzilla is perhaps the most popular one. When a system crashes, users can submit a bug or a crash report, which typically contains a description of the nature of the crash, system and platform information (i.e., OS, version, failed components, *etc.*), and stack traces. Major software companies such as Microsoft, Mozilla, and Apple, have deployed crash report feedback mechanisms on their software. They typically receive a large number of crash reports over an extended period of time and use them to guide their debugging efforts.

However, processing these incoming bug reports (BRs) can be tedious and time-consuming. Fortunately, not all crashes are caused by new bugs; many of them are duplicates of existing crashes. Anvik et al. [1] reported that approximately 20% of BRs for Eclipse and 30% of BRs for Firefox are duplicates. Lazar et al. [2] showed that the total number of duplicate BRs in Eclipse, OpenOffice, Mozilla, and Netbeans is up to 23% of the total reports. Due to a large number of bug reports submitted every day, the bug handling process tends to be challenging and time-consuming. Identifying duplicate BRs at an early stage can help improve the productivity of triaging teams, which in turn should speed up the bug resolution process. It is therefore essential to invest in techniques that can automatically detect duplicate BRs.

There exist studies to automatically detect duplicate BRs [3]–[6]. Most of them focus on the analysis of BR categorical data (e.g., component, product, *etc.*) and textual information such as BR descriptions and comments using machine learning and information retrieval techniques [4], [7]–[9]. Textual descriptions are written by end users, testers or developers may provide essential information for further analysis of BRs. However, BRs vary in their quality of content. They often contain incomplete or even incorrect

*Corresponding author:
E-mail addresses: n_ebr@ece.concordia.ca (N. Ebrahimi), trabelsi@ece.concordia.ca (A. Trabelsi), mdsha_i@ece.concordia.ca (Md. S. Islam), abdelw@ece.concordia.ca (A. Hamou-Lhadj), k_khanm@ece.concordia.ca (K. Khanmohammadi).

information [10], causing delays in fixing bugs due to poorly written described BRs. To overcome these limitations, researchers have turned to stack traces as the main features for detection of duplicate BRs [6], [7], [10]–[17]. Considered as an alternative and useful way to characterize a BR, a stack trace contains a sequence of running methods and threads in the system at the time of the crash. (Annotation 2.11) Stack traces have been used to help diagnose the causes of failures [18] or for bug reproduction [19], [20]. This is because they tend to be a more formal source of information than BR descriptions and comments that are entered by end users (including developers) using natural language. Stack traces are a useful alternative, especially when BR descriptions and comments suffer from quality problems due to noise in the data and the ambiguity and imprecision associated with the use of natural language.

In our previous work [21], we presented an approach for detecting duplicate BRs using stack traces and Hidden Markov Models (HMMs). An HMM is designed for the analysis of sequential data. This paper complements our previous work, which serves as a motivation for this work. More specifically, in this paper, we propose an approach that not only detects duplicate bug reports but also automatically assigns an incoming bug report to an appropriate and small-sized group of previously reported duplicate bug reports. We also fine-tuned and evaluated our approach on two new datasets, collected from Firefox and GNOME bug repositories, two large open source projects.

The proposed approach consists of four main steps (see Figure 2). First, we extract stack traces from each BR and, we split the resulting execution traces into duplicate groups of appropriate sizes. Then, we build and train an HMM model for every duplicate group. The generated HMMs are used to classify incoming BRs. Finally, the stack trace of each incoming BR is compared with the trained HMMs and classified according to the generated scores. We assessed the performance of our approach on BRs from Firefox and GNOME datasets by computing the recall@rank-k, where k ranges from 1 to 20, and the Mean Average Precision (MAP).

The main contributions of this paper compared to the previous one are:

- We experiment with new datasets, an updated Firefox dataset, and a new dataset of BRs of the GNOME system
- We provide a better evaluation of our approach using MAP and Recall@rank-k by varying k from 1 to 20. In the previous paper, we only presented precision and recall of the best result. Ranks provide a better view of the performance of our approach since it gives more flexibility to triagers and developers to search into a list of possible candidate duplicated BR groups.
- We compare our approach to the approach of Kim et al. [17], which uses graph theory to model stack traces for the detection of duplicates BRs.
- We provide a better discussion of our approach along with its limitations.
- We also improve the conclusion and future work section based on the lessons learned from this extended work.

The remainder of the paper is structured as follows. The next section surveys state of the art in duplicate-bug-report detection. In Section 2, we provide background information. Section 3 describes our proposed approach. Experimental results are reported in Section 4, followed with threats to validity in Section 5. Related work is presented in Section 6. We conclude the paper and discuss future work in Section 7.

2. Background

2.1. Bugzilla Bug Report Lifecycle

The process of entering and resolving a bug has a life cycle. A bug report may include fields such as ID, Product, Component, Assignee, Status, Resolution, Summary and Changed date and time. The lifecycle of a BR in Bugzilla is shown in Figure 1 [22].

At each stage, the status of a bug changes until a final resolution is found. A new bug is in the UNCONFIRMED status until its presence is confirmed. It remains in the NEW status until it is assigned to a developer. The status can be one of FIXED, DUPLICATE, WONTFIX, WORKSFORME or INVALID. The status is changed to RESOLVED FIXED if a developer fixes the bug. A WORKSFORME status means that the bug cannot be reproduced after some attempts by the developer.

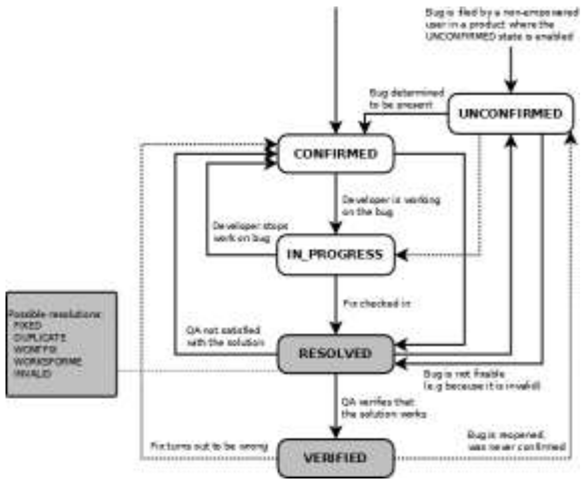
Bug is filed by a non-empowered user in a product where the UNCONFIRMED state is enabled
Bug determined to be present
UNCONFIRMED
CONFIRMED
Developer is working on the bug
Developer stops work on bug
IN_PROGRESS
Fix checked in
QA not satisfied with the solution
RESOLVED
Bug is not fixable (e.g because it is invalid)
QA verifies that the solution works
Bug is reopened, was never confirmed
Fix turns out to be wrong
VERIFIED
Possible resolutions:
FIXED
DUPLICATE
WONTFIX
WORKSFORME
INVALID

Fig. 1. Lifecycle of a Bugzilla Bug [22]

A bug is reopened if the tester is not satisfied with the fix or a formerly resolved report may be reopened at a later date (e.g., due to an ineffective fix). In such cases, the status of the BR changes to REOPENED [1], [12], [23]–[25]. Note that when a tester is satisfied with the fix, the status changes to VERIFIED. Finally, if the bug does not occur again the status changes to CLOSED [26].

2.2. Hidden Markov Models (HMMs)

An HMM is a statistical Markov model widely used to analyze the behavior of a system over time [27]. A more straightforward Markov process typically assumes that the states are directly visible to the observation data produced in the system. While in HMM, the states are hidden, but the output of each hidden state (i.e., the state transition probability) is dependent on the observation data. Moreover, based on the data types, an HMM can be further classified into discrete (typically the data is a discrete sequence produced from a finite number of tokens or symbols over time) or continuous (typically the data is generated from a Gaussian distribution such as speech, music, *etc.*). Since the observation data in our system is a discrete sequence of function calls, we have used the discrete form for the output distributions to model the function calls forming stack traces. A typical topology of an HMM is shown in Figure 2.

Training an HMM model requires defining the following parameters:

Number of hidden states: To train an HMM model, we need to set the number of hidden states (N) in the Markov process. Let the distinct states in a Markov process be $S_i,\ i = \{0,1,\ldots,N-1\}$ and the notation $X_t = S_i$ represents the hidden states sequence S_i at time t.

Number of observation symbols: To train an HMM model, we need to set the number of observation symbols (M). Let the distinct observation symbols be $R_k,\ k = \{0,1,\ldots,M-1\}$ and the notation $O_t = R_k$ represents the observed symbol R_k at time t for the given sequence of observations $(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$, where T is the length of that sequence.

State transition probability distribution: The first-row stochastic process is the hidden state transition probability distribution matrix $A = \{a_{ij}\}$. A is an $N \times N$ square matrix and the probability of each element $\{a_{ij}\}$ is calculated by the following equation:

$$a_{ij} = P\big(state\ S_j\ at\ t+1 \big| state\ S_i\ at\ t\big),\ i,j = \{0,1,\ldots,N-1\} \qquad (1)$$

In Equation (1), the transition from one state to the next is a Markov process of order one [27]. This means that the next state depends only on the current state and its probability value. Since the original states are "hidden", we cannot directly compute the probability values in the past. However, we can observe the observation symbols for the current state S_i at time t from the given observations sequence $\mathcal{O}$ to train an HMM model.

Observation symbol probability distribution: The second-row stochastic process is the observations symbol probability distribution matrix $B = \{b_j(R_k)\}$. B is an $N \times M$ matrix which is computed based on the observation sequences (i.e., the temporal order of stack traces). The probability of each element $b_j(R_k)$ is given by the following equation:

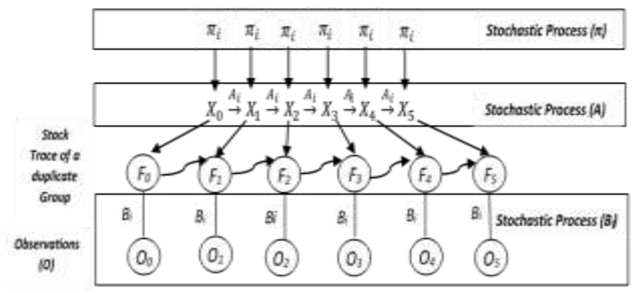

Fig. 2. Typical topology of an HMM.

$$b_j(R_k) = P\big(observation\ symbol\ R_k\ at\ t \big| state\ S_j\ at\ t\big) \qquad (2)$$

Initial state probability distribution: The third-row stochastic process is the initial state probability distribution $\pi = \{\pi_i\}$. π is a $1\times N$ row matrix and the probability of each element $\{\pi_j\}$ is given by Equation (3):

$$\pi_i = P(state\ S_i\ at\ t = 0) \qquad (3)$$

Training an HMM model aims to maximize the likelihood function $P(\mathcal{O} \mid \lambda)$ over the above three-parameter space. The Baum-Welch (BW) algorithm is the most commonly employed expectation-maximization (EM) algorithm to estimate HMM parameters. It uses a forward-backward (FB) algorithm at each iteration to efficiently evaluate the likelihood function $P(\mathcal{O}|\lambda)$. It updates the model parameters till a maximum number of iterations is reached or the likelihood function achieves no more improvement. In this paper, we also use BW to estimate HMM parameters.

3. Experimental Setup

3.1. Overall Approach

Figure 3 shows our approach, which consists of the following steps. First, we extract BRs with stack traces from BR repositories of Firefox and GNOME, which use Bugzilla for BR tracking. For Firefox, there is another system that manages crash reports, which contains stack traces. The link between a BR in Bugzilla and its corresponding crash reports in Mozilla, if it exists, is established through the bug ID (see Section 3.2 for more details). For GNOME, traces are embedded within the BRs. Once we have the BRs, we search for the duplicates one by examining the BR status. We create duplicate BR groups (DG) where each group DG_i contains stack traces of one master BR and those of all its duplicates. We have as many duplicate groups as master BRs. For each DG_i, we train an HMM using 60% of the traces, validate the HMM using 10% of the traces, and test the model using 30% of the traces of this DGi and every other DG. These steps are discussed in more details in the next subsections.

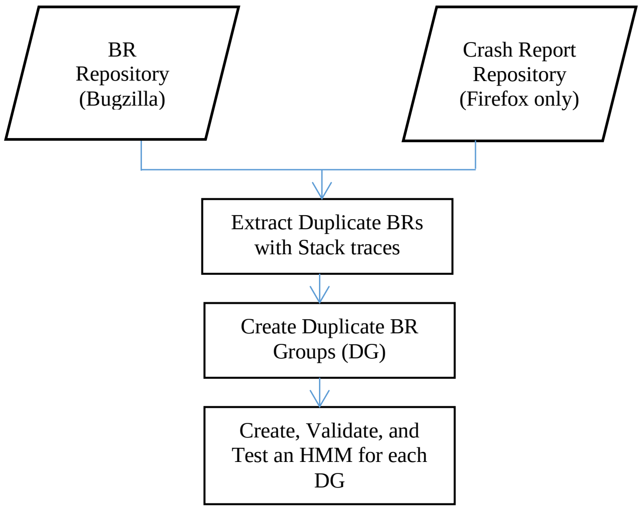
BR Repository (Bugzilla)
Crash Report Repository (Firefox only)
Extract Duplicate BRs with Stack traces
Create Duplicate BR Groups (DG)
Create, Validate, and Test an HMM for each DG

Fig. 3. Overall approach

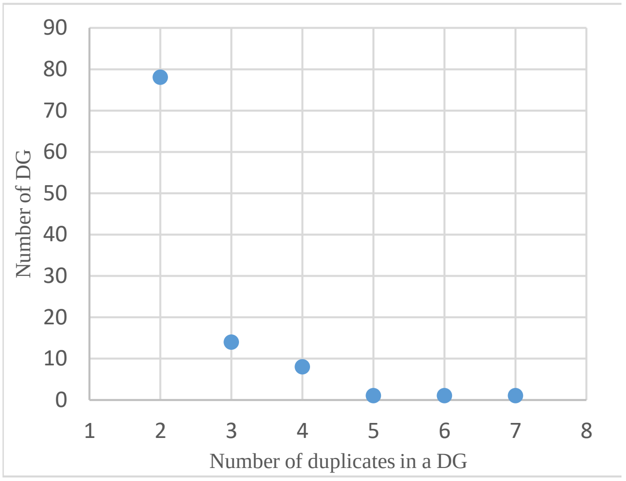

Fig. 4. Number of duplicate BRs in each DG in the Firefox dataset.

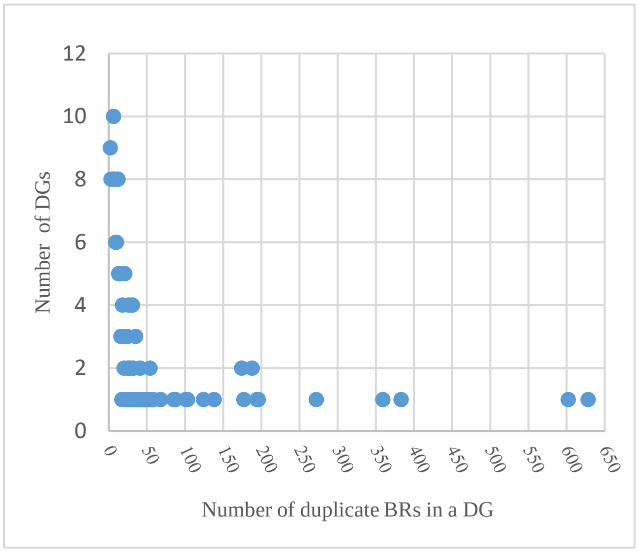

Fig. 5. Number of duplicate BRs in each DG in the GNOME dataset.

3.2. Datasets and formation of duplicate BR groups

Our empirical study was performed on bug reports extracted from the two large open source software projects: Firefox and GNOME.

Firefox dataset: Firefox is a well-known Internet browser used by millions of users around the world. Firefox uses Mozilla as a crash reporting system[1] (more precisely, the Mozilla crash reporting system is called Socorro), and Bugzilla for tracking BRs. There is a difference between a crash report and bug report for Firefox and other Mozilla products. A crash report is automatically reported when a crash occurs. The report contains various fields such as the stack trace, priority, product, component, and OS version. Similar crash reports are grouped into clusters according to their top-method signature and a clustering algorithm [28]. When the number of crash reports in the same cluster reaches a certain threshold, a BR is created and submitted to Mozilla's bug tracking system, Bugzilla. This is usually done manually. (Annotation R2.13) Most BRs do not contain stack traces, but contain links to signatures that refer to groups of crash reports. The link, if it exists, is through a BR ID.

We implemented a web crawler to extract duplicate BRs in the Bugzilla website from BR#1 to BR#1,299,999. Bugzilla uses the "Product" and "Component" to classify bugs. We used mainly the BRs associated with the "Core" Product, which contains bugs in components used by Firefox and other Mozilla software. We extracted BRs with status "RESOLVED DUPLICATE" or "VERIFIED DUPLICATE" and found their corresponding master BRs. (Annotation R2.14) In addition, we only included in the dataset BRs that have links to their corresponding crash reports since our approach uses stack traces, which are only kept in crash reports, i.e., not copied to BRs.

[1]https://crash-stats.mozilla.com/home/product/Firefox

We create a duplicate BR group (DG) by including stack traces of the master BR and those of its duplicates. For example, the master BR Bug#1236639 was reported in January 2016 and later Bug#1258802, Bug#1260779 and Bug#1263241 were reported in 2016 and marked as duplicates of Bug#1236639. The crash traces associated with all these BRs that are retrieved from Mozilla are put together to form a duplicate BR group.

We only included duplicate BRs that have at least four stack traces, the strict minimum number of traces needed to construct the training, validation, and testing sets. (Annotation 2.16) It should also be noted that we removed functions that contain "0x" (e.g., windows.dll@0x), and the ones that start with "Fxxxx". These functions exist when debugging information could not be used to identify the function names associated with the running function in a memory dump. This may be caused by several reasons including crashes that make the program jump to a random address in memory, third-party DLLs, obfuscation, *etc*. We also removed repetitions of the exact same traces.

In addition, in Mozilla, crash reports may be associated with more than one crashing thread starting from Thread 0. We have implemented a crawler to retrieve all available threads in a crash report. For both convenience and computation time reduction, we have only kept Thread 0 into consideration. We also only limited the number of crash traces associated with each bug report to 200 stack traces.

The total number of duplicate bug report groups (DGs) is 103. Figure 4 and Table 1 show the number of DGs for Firefox. About 73% (90 DGs) contain only two duplicate BRs. Others contain duplicates ranging from 2 to a maximum of 8. While only 5 out of 103 DGs include more than 5 duplicates.

GNOME: GNOME is a graphical user interface and a set of desktop applications for Linux. It also uses Bugzilla bug tracker [26] to track bug reports. Unlike Firefox that uses a different system to track stack traces, GNOME's BRs have stack traces as part of the BR description. We implemented a web crawler to retrieve all GNOME BRs from Bugzilla. We collected all available GNOME BRs that were reported until December 2016, which represents a total of 753,300 BRs.

Similar to Firefox, we created DGs by putting in each DG stack traces of a master BR and those of its duplicates. We only kept duplicate BRs with a minimum of four stack traces to construct training, validation, and testing sets. We preprocessed GNOME traces by deleting calls to Java libraries or setting up of debugging parameters. We also removed repetitions of the exact same traces as we did for Firefox.

We formed 182 DGs. Fig. 6 and Table 1 show the characteristics of the GNOME DGs. A wide range of duplicates ranging from 2 to 628 are available in each duplicate group. As can be observed from Figure 5, 156 out of 182 (about 86%) of DGs contain up to 50 duplicates. Comparing both datasets, the GNOME dataset is larger in the sense that it contains more duplicate groups and each duplicate group contains a large number of traces.

Table 1
Characteristics of the duplicate BRs groups of Firefox and GNOME datasets

Dataset	Total number of duplicate BR groups	Total number of traces
FIREFOX	103	2,883
GNOME	182	4,600

3.3. Training Phase

We build an HMM for each duplicate BR Group, DG_i. We use 60% of traces in DG_i for training. We vary the number of hidden states N from 15 to 50 with a leap out of 5. To our knowledge, no study specifies how to set the number of hidden states. Most studies that use HMMs set this parameter through experimentation. In our case, we found that for Firefox and GNOME, the best accuracy is obtained when N=20 and N=40, respectively. We experimented with higher N > 50 values and observed no improvement.

3.4. Validation Phase

Validation is used to better estimate the best fit for the HMM parameters A, B, and π. In our study, we used 10% of traces in each DG_i to validate the HMM constructed through training. We performed 10 iterations[2] to estimate the HMM parameters *A*, *B*, and *π*. Initial parameter values are passed to the Baum-Welch algorithm to compute the log-likelihood as scores for all traces inside the validation set. The best-recorded parameters $A_i\,, B_i \; and \; \pi_i$ that are obtained from the minimum mean value among the 10 iterations are used to construct the HMM models.

3.5. Testing Phase

We used 30% of traces from each DG as a testing dataset. Let a new stack trace ST_i be mapped to sequences of observations $O_i = \{O_1, O_2, \ldots, O_{T-1}\}$. The latter ones are presented to HMM models, $\lambda_l = \{\lambda_1, \lambda_2, \ldots, \lambda_L\}$, of all DGs. Then, the log-likelihood of the sequence of observations $P(\,O_i|\lambda_l), \forall\, l = \{\,1, 2, 3, \ldots, L\}$ for every trained HMM model is calculated. A set of ordered scores for all HMM models, $S = \{S_1, S_2, \ldots, S_L\}$ is subsequently generated and reported to specify possible labels within the ranked list.

3.6. Evaluation Metrics

We used the recall rate and the Mean Average Precision (MAP) to assess the effectiveness of our approach. These metrics are used extensively in the literature [9], [29]–[32] so as to evaluate the performance of a ranked list. The recall@rank-k is defined as:

$$Recall@rank - k = \frac{N_{detected@k}}{N_{total}} \quad (4)$$

where $N_{detected@k}$ is the number of correctly retrieved k stack traces. Recall@rank-k is defined as the percentage of duplicates for which the master is found for a given top list size k.

The Mean Average Precision (MAP) indicates how accurately duplicate candidates are ranked. It is measured as follows:

$$MAP = \frac{1}{Q}\sum_{n=1}^{Q}\frac{1}{rank(n)} \quad (5)$$

[2]Our experiments have shown that after 10 iterations, the parameter values do not vary.

where Q is the number of correctly retrieved duplicate candidates and $rank(n)$ is the position in which the right stack trace is retrieved.

MAP ranges from 0 to 100%. An approach that returns MAP=100% means that for all BRs in the testing set, the approach was able to classify them accurately at the top rank. It is sufficient for a triager to look at one DG to find the corresponding duplicate group. A MAP close to zero means that the approach would return many possible DGs for each BR in the testing set because of a poor classification.

4. Evaluation

4.1. Firefox and GNOME Datasets

The results of applying our approach to Firefox dataset are shown in in Table B1 (see Appendix B). The recall rates with ranks with k ranging from 1 to 20 show that our approach achieves promising results across all HMMs with different states. The average recall for Rank k =1 is 59%, for Rank k=2 is 75.55%. We start reaching the 90% recall from k=10.

In the case of MAP (see Figure 6), we obtained MAP values between 75.77% and 76.44% with different numbers of hidden states, an average of 76.24%. In other words, a given incoming BR can be identified by our approach in the first DG that the approach suggests with 76% of chances. We pass to the 85% MAP bar with 5 DGs, which we believe it is considered a good result.

Table B2 shows the results of the recall at rank k with k ranging from 1 to 20 with HMM models with different state numbers. The recall at k=1 is almost 63% for all state numbers, while this value increases by about 10% for k=2. It can also be observed that having a recommended list of 2 duplicate groups, detection accuracy of about 73% can be achieved. This detection accuracy increases to 97% for k=11. In terms of MAP rate, values of about 72% and 73% were achieved using state numbers of 40 and 45, respectively (see Figure 7). Similar to Firefox, we did not see a significant impact in changing the number of hidden states.

We also found that changing the number of hidden states does not substantially impact the recall rate in the ranked list as shown by the boxplots of Figures A1 and A2 for both Firefox and GNOME.

4.2. Comparison

In this section, we compare our approach to the approach proposed by Kim et al. [17], which models aggregated views

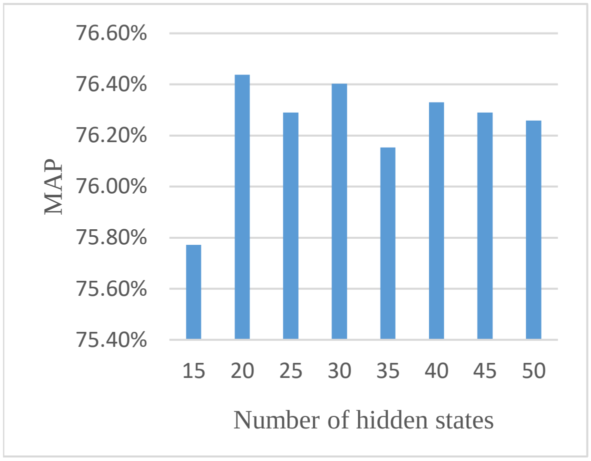

Fig. 6. MAP obtained with different HMM state numbers for Firefox

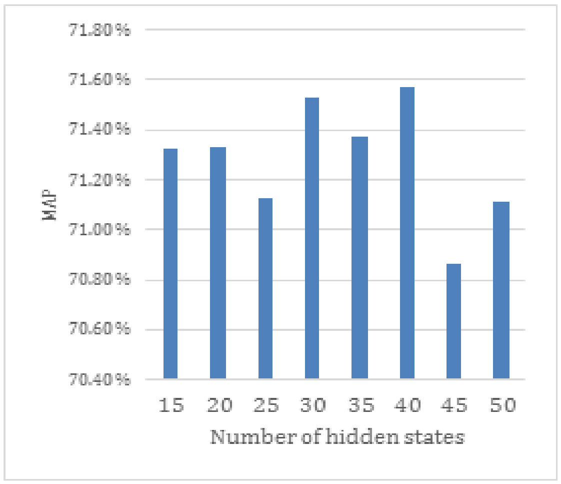

Fig. 7. MAP obtained with different HMM state numbers for Gnome

of crash traces using graphs to detect duplicate crash reports in the Windows Error Reporting (WER) system in order to facilitate triaging tasks.

The approach, which we refer to as CrashGraph in this paper, aggregates multiple stack traces (called crash traces in [17] in the same group by constructing a graph where the nodes represent the stack trace functions and the edges represent the calling relationship.

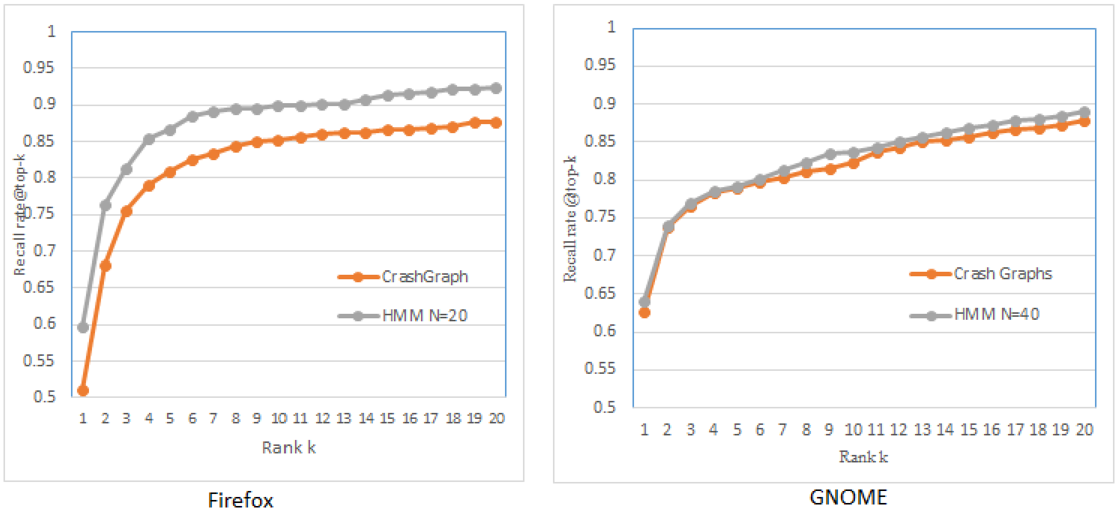

Fig. 8. Comparison of Recall rate@rank-k between CrashGraph and HMM

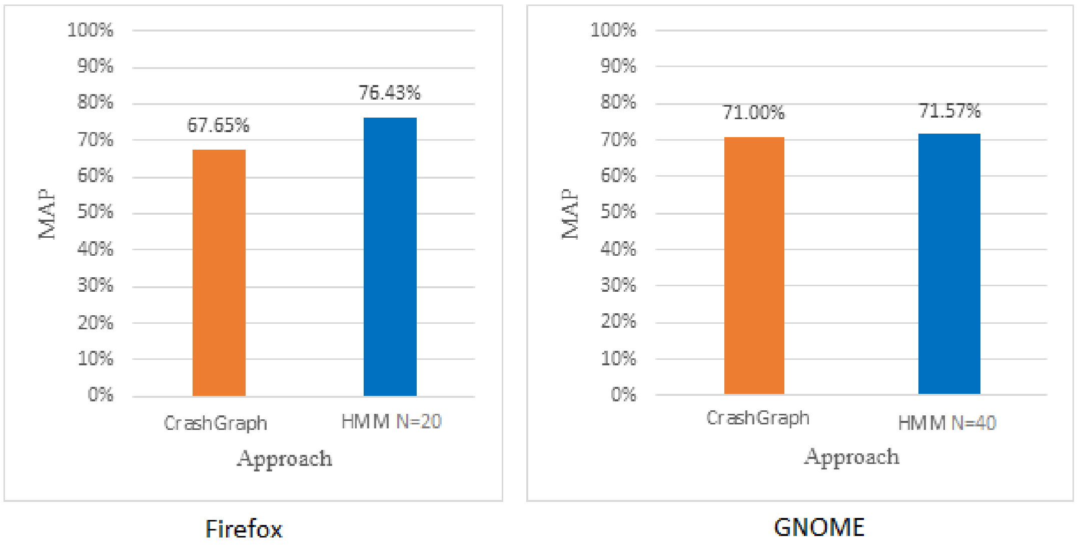

Fig. 9 Comparison of Recall rate@rank-k between CrashGraph and HMM

Figure 10, taken from [17], shows an example of three stack traces ABCD, AFGD, and CDFG, where A, B, C, D, F, G are functions. In this example, a graph is created by taking a 2-gram representation of trace elements and combining them.

The authors use the aggregated graph to model stack traces of each bucket (a group of related crash reports in WER) to predict if a new crash trace should belong to the same bucket or not. A similarity metric is used to determine the extent to which an incoming stack trace is deemed similar to those modeled in the graph. When applied to detect duplicate bug reports in two Windows products, the best accuracy achieved by CrashGraph is 71.5% precision and 62.4% recall using a 99% similarity, which we also used to run the experiments with CrashGraph.

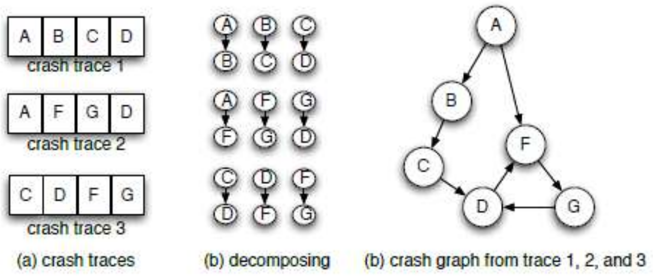

Fig. 10. An example of a graph created by CrashGraph to model traces [17]

We implemented CrashGraph to the best of our knowledge. We applied it to DGs of Firefox and GNOME to predict the DG of an incoming stack trace. We used the same setting as before. More precisely, for each DG$_i$ (whether it is for Firefox of GNOME), we used 60% of traces to construct a CrashCraph and 20% of traces to test it. Note that we did not use the validation set to validate CrashGraph since CrashGraph does not use any particular heuristics.

We compared CrashGraph with HMM N=20 for Firefox, and HMM N=40 for GNOME since these are the HMM models that provide the best accuracy. The results are shown in Figures 8 and 9.

The results show that HMM performs better than CrashGraph when applied to the Firefox dataset. At Rank 1, our approach achieves a recall of 59.7% whereas CrashGraph achieves a recall of 51%. This gap is maintained as k increases as shown in Fig. 9. MAP of HMM is also better than the one obtained with CrashGraph.

For GNOME, our HMM-based approach performs almost the same as CrashGraph as shown in Figures 8 and 9. This may be due to the fact that there are many more traces in GNOME than Firefox. CrashGraph was able to build a representative graph that characterizes the traces of a DG.

4.3. Discussion and Limitations

A. Varying the number of hidden states

To train the HMM models, we varied the number of hidden states from 15 to 50 with bounds of 5. A different setting may lead to different results. However, our results suggest that the number of hidden states does not have major impact on the overall approach. As we can see from the boxplots in Figures A1 and A2 (see Appendix A), the recall changes slightly by varying the number of hidden states. Take for example the results obtained for Rank 1 for Firefox, the recall ranges from 57.91% to 59.81%, i.e., a 1.9% difference.

B Impact on triaging effort

The MAP is an indication of how well a given approach ranks incoming BRs (in our case a BR is characterized by its stack trace). The average MAP across all HMMs is 76% for Firefox and 71% for GNOME. This means that, in general, our approach ranks well the incoming BRs, which should reduce the time spent by triagers to find the right DG for an incoming BR. The better the recall and MAP, the less effort is needed. Note that for both datasets, the gap between recall at Rank 1 and Rank 2 is significantly reduced when comparing recalls between the subsequent pairs of ranks (Rank 2 with Rank 3, Rank 3 with Rank 4, *etc.*). This suggests that MAP should be even higher if we ignore Rank 1, meaning that a triager would accept to examine at least two DGs to determine the right DG.

C Differences between Firefox and GNOME

We found that our approach (as well as the CrashGraph approach) performs better for GNOME than for Firefox. This may be due to the fact that GNOME has more traces than Firefox (4,600 compared to 2,883). More traces in a DG mean a better characterization of BRs of the same group, which helps with the classification process achieved by HMM (and CrashGraph). We also have more DGs in GNOME than in Firefox. Besides, GNOME has more duplicates in each DG than Firefox as shown in Figures 4 and 5. This may be due to the fact that crash reports go directly in the bug reporting system in GNOME, which is not the case for Firefox. There is an additional triage phase that Mozilla does between the report of crashes and the filing of bugs, which is not done in GNOME, resulting in less duplicates BRs for Firefox than GNOME.

D. Limitations

One of the main limitations of our approach is that it does not deal with new incoming BRs that do not have prior duplicates in the database. In other words, we extend the approach to consider the creation of new groups on the fly. One possibility to do this is to determine a threshold below which an incoming BR is deemed dissimilar enough to all existing BRs and hence should be put in a new group that needs to be created. This threshold can be determined during the validation step of our approach using a validation set that contains a mix of duplicate BRs and new BRs (BRs without duplicates).

Another limitation of our approach is due to the low number of BRs with stack traces in existing BR repositories. Unfortunately, not all BR tracking systems collect traces automatically, making it difficult for a user to submit traces. As an example, only 10% of Eclipse bug reports described by Lerch et al. [33] contain stack traces. Mozilla keeps stack traces for only one year because of the cost of saving a large number of stack traces. Nevertheless, we believe that an approach that uses stack traces remains very useful, especially because stack traces are needed for other tasks such as bug reproduction [19], [20]. We conjecture that, in the future, more bug tracking systems will collect automatically stack traces and use advanced storage mechanisms to make traces available for more extended periods of time.

5. Threats to Validity

Our proposed approach and the conducted experiments are subject to threats to validity, namely external, internal, and construct validity.

5.1. Threats to external validity

Our approach is evaluated against two open source datasets and tested on duplicate BRs with stack traces. We need to apply our technique to more datasets. We also need to evaluate if it outperforms existing work and approaches that use other BR features such BR descriptions and comments.

5.2. Threats to internal validity

In our approach, the way we set the parameters A and B, and the conditional probability matrices to construct HMMs could be a threat to internal validity. We used the validation set to set the bounds to optimize A and B. A different validation set could result in a different initialization, resulting in a different model. However, to our knowledge, there is no clear solution to this problem and most studies that use HMM follow random initialization of A and B and repeat this process several times until a satisfactory model is obtained.

The way we set the number of hidden states is another threat to validity. We followed the common practice of setting this number to a small number and then increase it with bounds of 5. Although different state numbers do not seem to bring much improvement in accuracy (see Figure A1 in Appendix A), there is always a possibility that a different number may lead to other models.

For Firefox, we used traces of Thread0 only. Considering all threads will require an extensive amount of time to train each HMM. However, for crashes due to hangs, the top few methods are more or less the same (with functions such as "wait") between crash reports even if the root causes of the crashes are different. This may cause the associated BRs to end up in the same bug report groups. To address this issue, we should (a) examine in depth what the impact would be, and (b) consider including other threads. This said, considering all threads may cause scalability problems. Therefore, a trade-off between precision and scalability should be investigated.

The Firefox results depend on the quality of the initial manual triage and the quality of the signatures. Errors in manual triage and incorrect signatures may affect our results.

Finally, another threat to internal validity is related to the web crawling and parsing tools implemented to collect BRs and extract stack traces, and in the way we implemented the CrashGraph algorithm. To mitigate this threat, we verified our data multiple times. We also intend to release a reproduction package to allow other researchers to reproduce our research work.

5.3. Threats to construct validity

The construct validity shows how the used evaluation measures could reflect the performance of our predictive model. In this study, we used recall and Mean Average Precision, which are widely used in other studies to assess the accuracy of machine learning models with applications to the problem of duplicate BR detection.

6. Related Work

Several approaches have been proposed in the literature to support the automatic detection of duplicate bug reports. This section presents previous studies related to our work. They can be divided into two main categories based on the type of BR information used: 1) Textual-based approaches and 2) Execution information-based approaches.

6.1. Textual-based approaches

Typically, developers and users submit information related to the crash in the summary and textual description part of a bug report. The aforementioned unstructured information can then be used by researchers to investigate the similarities between existing bug reports and an incoming bug report for classification purposes [4], [9], [19], [29], [31], [34]–[42]. Using natural language information, information retrieval (IR) techniques are widely used to calculate the similarity scores between queries and the retrieved data. In this context, Wang et al. [36] have studied similar terms in duplicate bug reports of OpenOffice for text similarity measurements. Then, they proposed a new technique to extract similar terms in BR descriptions that cannot be obtained via a general purpose thesaurus. They showed that their method improves existing methods by 58%. Rakha et al. [7] have conducted a study on the effort required for manually identifying duplicate issue reports for Firefox, SeaMonkey, Bugzilla, and Eclipse-Platform. They have observed that more than 50% of duplicate bug reports can be detected within 24 hours after their submission even when only one developer is involved. Their classification model achieves an average precision and recall of 68% and 60%.

The performance of different IR models (e.g., Log-Entropy based weighting systems) compared with topic-based modes (e.g., LSI, LDA, and Random Projections) has been studied by Kaushik et al. [34]. By applying different heuristics on data retrieved from Eclipse and Firefox, they have observed that word-based models outperform topic-based models with 60% and 58% recall rates, respectively. Their results suggest that the project's domain and characteristics play a crucial role in improving the performance of heuristic models.

Sun et al. [32] proposed a supervised approach based on a discriminative model. The model uses information retrieval to extract textual features from both duplicate and non-duplicate bug reports. It is then trained and tested using a support vector machine (SVM) classifier. All pairs of duplicate bug reports have been formed and considered as positive samples, while all other pairs of non-duplicate bug reports have been treated as negative ones. By applying the method on BRs from Firefox, Eclipse, and OpenOffice, the authors' method was able to achieve a recall as high as 65% on all datasets.

In another studies [32], the authors have improved their model by extending the well-known BM25 ranker to provide a ranked list of duplicates. It should be noted that BM25 is a function for calculating term frequencies and similarities among bug reports. By taking advantage of IR-based features and topic-based features, Nguyen et al. have extended the work of Sun et al. by combining BM25F with a specialized topic model. The authors have considered words and term occurrences inside bug reports in an effort to improve bug localization performance. The algorithm shows recall rate@k between 37% and 71% and MAP of 47%.

Sureka and Jalote [40] proposed a character-level n-gram approach to further improve the accuracy of automatic duplicate-bug-report detection. The technique calculates the text similarity between the user's query and existing title and description information of bug reports in character-level. The character-level n-grams are language independent and thus, they save languages specific pre-processing time. According to their experiments, however, their approach has been of modest performance for which about 21% and 34% recall rates have been achieved for top 10 and top 50 recommendations, respectively.

Another set of supervised approaches build a model based on a training data and use it to analyze a pair of BRs to predict whether they are duplicate. In addition to textual and categorical features (description, component, priority, *etc.*), Alipour et al. [29] suggested using contextual features to detect duplicate BR pairs. They showed that domain knowledge of software engineering concept plays a compelling role in detecting duplicates between BR pairs. When applied to a bug repository of the Android ecosystem, the approach achieves a recall of up to 92%.

Deshmukh et al. [3] applied Convolutional Neural Networks (CNN), and Long Short Term Memory (LSTM) on short and long descriptions of BRs extracted from Lazar et al. [2] dataset. They showed that their approach could achieve accuracy and recall rate of 90% and 80%, respectively.

6.2. Execution information-based approaches

Wang et al. [39] applied natural language processing techniques on both stack traces and BR descriptions and observed that there is an improvement of 25% over approaches that only use BR descriptions. The authors, however, did not model the temporal order of sequence of calls in stack traces. Instead, they treated stack traces as text with stack trace functions are treated as words. This approach detects 67%-93% of duplicate BRs of Firefox.

Lerch et al. [33] proposed an approach to identify stack traces in BRs by transforming stack traces into a set of methods and then using term frequency to compute and rank the similarity between method sets. The authors' method, when applied to Eclipse BRs, achieves the same results as the state-of-the-art approaches, but with fewer requirements. This approach, however, does not take into account the temporal order of sequences of function calls in stack traces.

Kim et al. [17] proposed a crash graph-based model which captures the crashes reported and stored in a bucket. A graph of stack traces in a bucket (a group of related bug reports) is constructed to aggregate multiple traces. Instead of comparing an upcoming stack trace with every single trace in a bucket, their model only compares with the graph. To evaluate their model, the authors used graph similarity as a metric. When applied to crash reports of Windows systems, their approach achieves a maximum precision of 71.5% and recall of 64.2%. To our knowledge, this is the only approach that uses temporal order of sequences of functions calls of stack traces to detect duplicates. Our approach achieved a better recall rate and MAP than Kim et al.'s approach as discussed in Section 4.3.

To improve the accuracy of bucketing in the Windows Error Reporting system (WER), Rebucket was proposed by Dang et al. [5] for clustering crash reports based on call stack similarity. Rebucket measures the similarity between call stacks in WER and assigns crash reports to buckets according to similarity values. This approach is not used to detect duplicates, but group related crashes together.

In our previous work [13], we collected stack traces in a small group of duplicates with a varying length n-grams and automata (a Markov model). The automaton has been taken as a representative for the duplicate group and test stack traces of that duplicate group. When an incoming stack trace is sufficiently similar to the automaton of a duplicate group, it is labeled as a duplicate of that similar group. In our paper [21], we improved CrashAutomata using HMMs.

Sabor et al. [43] used package names in stack traces instead of method names to detect duplicates BRs in Eclipse. Their method then generates n-gram features from sequences of package names. The extracted features are then used for measuring the similarity between new stack traces of new and stack traces of historical BRs. The objective of their paper is to reduce the computation time to process large traces.

Castelluccio et al. [28] proposed a tool, which was integrated in Socorro, to find statistically significant properties in groups of Mozilla crash reports and present them to analysts (developers and triaging teams) to help them analyze the crashes and understand the causes. The tool is based on contrast-set learning, a data mining approach [44]. The authors applied the tool crash data collected from the Mozilla crash reporting system and bug tracking system. Their findings show that the tool is very effective in analyzing related crash groups and bugs.

Furthermore, the tool, which is now integrated with the Mozilla crash reporting system, received favorable feedback from Mozilla developers. Although this tool does not tackle the problem of duplicate BR reports, it could be useful in our research. We can use it to extract the most meaningful properties of crash traces and use them to improve the HMM models. The tool can also be used to improve the grouping of crashes in Mozilla, and use the resulting grouping to identify crash reports that are related to duplicate BRs.

6.3. Discussion

As described in the previous section, the majority of studies focus on detecting duplicate BRs using the textual parts of BRs such as summary and description. Most of the techniques that use stack traces treat their content as document and leverage natural language processing techniques. These techniques do not take advantage of the temporal order of function calls in stack traces, which characterize the execution of a system. We conjecture in this paper that they are an excellent alternative to BR descriptions, especially in cases where we cannot rely on BR descriptions and comments because of the quality issues.

To our knowledge, the only technique that truly leverages stack traces for the problem of duplicate BR detection is the one proposed by Kim et al. [17] by leveraging graph theory techniques. As shown in the comparison section, our approach performs better than the approach proposed by Kim et al.

7. Conclusion and Future Work

In this paper, we have presented a novel approach aimed at automatically detecting duplicate bug reports using execution traces and Hidden Markov Models. Based on our study, we recognize the obvious benefits we derive from using stack trace's information solely that we believe improves the detection accuracy of duplicate bug reports.

Our experiments highlight that with a list of rank-1 bug reports, recall values of 80% and 63% have been achieved on Firefox and GNOME datasets, respectively. With the same list of bug reports, our approach detects the duplication of a given report with an average MAP value of 87% and 71.5% on Firefox and GNOME datasets, respectively. It has also been observed that the higher the rank level, the higher the recall rate. For instance, the recall rate with a list of rank-2 has been about 12% higher than that with a list of rank-1.

In the future, we plan to investigate more BRs from additional software systems. We also plan to improve the effectiveness of our proposed approach in terms of recall and MAP scores.

In addition, for Firefox, we used only traces coming from Thread 0, we need to extend the dataset by considering more threads. This way, we can have a better characterization of a BR by using traces from multiple threads.

Also, we will study how we can combine stack traces with other BR fields such as BR descriptions and comments. A combined approach should not treat stack traces as documents, as it is done in the literature, but model the temporal order of sequences of function calls, just as it is done in this paper.

Another interesting future work direction is to investigate the use of our HMM-based approach to detect similar crash reports in Mozilla and compare it to the baseline solution of duplicating crash reports based on method signatures. If effective, our approach can be used on top of Socorro's algorithms for detecting similar crash reports [28].

Finally, we stress the need to examine the efficiency and scalability of our approach by measuring the execution time and other related factors. This is needed when HMMs are used because HMM is known to cause scalability problems when applied to very large datasets. The problem is not only for the initial training of the model but also for model updates, i.e., when new traces (BRs) should be added to the model.

Acknowledgement

We would like to thank the anonymous reviewers for their excellent comments and feedback. We would also like to thank NSERC (Natural Science and Engineering Research Council of Canada) for partly supporting this projet.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," *Proc. of OOPSLA Work. on Eclipse Technol. Exch. eclipse'05, Oct. 16, 2005 - Oct. 17, 2005*, pp. 35–39, 2005.

[2] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," *Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014*, pp. 392–395, 2014.

[3] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards Accurate Duplicate Bug Retrieval using Deep Learning Techniques," pp. 115–124, 2017.

[4] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR*, pp. 385–390, 2012.

[5] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," *Proc. of Int. Conf. Softw. Eng.*, pp. 1084–1093, 2012.

[6] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," *Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014*, pp. 308–311, 2014.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... Really?," *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 337–345, 2008.

[8] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 52–61, 2008.

[9] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," *2010 ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, pp. 45–54, 2010.

[10] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?," *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 618–643, 2010.

[11] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," *Proc. - Int. Conf. Softw. Eng.*, pp. 118–121, 2010.

[12] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," *Proceeding 28th Int. Conf. Softw. Eng. - ICSE '06*, vol. 2006, p. 361, 2006.

[13] N. Ebrahimi Koopaei and A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata," *Proc. 25th Annu. Int. Conf. Comput. Sci.*

Softw. Eng., pp. 201–210, 2015.

[14] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," 2014.

[15] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. I. Matsumoto, "Predicting re-opened bugs: A case study on the Eclipse project," *Proc. - Work. Conf. Reverse Eng. WCRE*, pp. 249–258, 2010.

[16] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014, pp. 151–160.

[17] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 486–493, 2011.

[18] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "CrashLocator: locating crashing faults based on crash stacks," *Proc. 2014 Int. Symp. Softw. Test. Anal. - ISSTA 2014*, pp. 204–214, 2014.

[19] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "A bug reproduction approach based on directed model checking and crash traces," *J. Softw. Evol. Process*, vol. 29, no. 3, p. e1789, 2017.

[20] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *PhD Thesis, Honk Kong Univ. Sci. Technol.*, vol. 41, no. 2, pp. 198–220, 2015.

[21] N. Ebrahimi Koopaei, M. S. Islam, A. Hamou-Lhadj, and M. Hamdaqa, "An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models," pp. 75–84, 2016.

[22] "Life Cycle of a Bug." [Online]. Available: https://www.bugzilla.org/docs/4.4/en/html/lifecycle.html. [Accessed: 23-Jul-2018].

[23] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?," *2010 ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, pp. 55–64, 2010.

[24] G. Cuevas, "Managing the software process with a software process improvement tool in a small enterprise," *J. Software-Evolution Process*, no. July 2010, pp. 481–491, 2012.

[25] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed ?," *Proc. 34th Int. Conf. Softw. Eng.*, pp. 14–24, 2012.

[26] "Bugzilla." [Online]. Available: https://bugzilla.mozilla.org/.

[27] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Ieee*, vol. 77. pp. 257–286, 1989.

[28] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, "Automatically analyzing groups of crashes for finding correlations," *Proc. 2017 11th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2017*, pp. 717–726, 2017.

[29] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," *IEEE Int. Work. Conf. Min. Softw. Repos.*, pp. 183–192, 2013.

[30] M. S. Rakha, C. P. Bezemer, and A. E. Hassan, "Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports," *IEEE Trans. Softw. Eng.*, vol. 5589, no. c, pp. 1–27, 2017.

[31] A. T. Nguyen, T. T. T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. - ASE 2012*, p. 70, 2012.

[32] C. Sun, D. Lo, S. C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," *2011 26th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2011, Proc.*, pp. 253–262, 2011.

[33] J. Lerch and M. Mezini, "Finding Duplicates of Your Yet Unwritten Bug Report," Proc. of Int. Cong. on Soft. Maintenance and Reeng. *CSMR*, pp. 69–78, 2013.

[34] N. Kaushik and L. Tahvildari, "A comparative study of the performance of IR models on duplicate bug detection," *Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR*, pp. 159–168, 2012.

[35] M. J. Lin, C. Z. Yang, C. Y. Lee, and C. C. Chen, "Enhancements for duplication detection in bug reports with manifold correlation features," *J. Syst. Softw.*, vol. 0, pp. 1–11, 2014.

[36] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting Paraphrases of Technical Terms from Noisy Parallel Software Corpora," *Proc. ACL-IJCNLP 2009 Conf. Short Pap.*, no. August, pp. 197–200, 2009.

[37] M. S. Rakha, W. Shang, and A. E. Hassan, "Studying the needed effort for identifying duplicate issues," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 1960–1989, 2016.

[38] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," *Proc. - Int. Conf. Softw. Eng.*, pp. 499–508, 2007.

[39] X. W. X. Wang, L. Z. L. Zhang, T. X. T. Xie, J. Anvik, and J. S. J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," *2008 ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008.

[40] A. Sureka and P. Jalote, "Detecting duplicate bug report using character N-gram-based features," *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 366–374, 2010.

[41] A. Tsuruda, Y. Manabe, and M. Aritsugi, "Can We Detect Bug Report Duplication with Unfinished Bug Reports ?," *2015 Asia-Pacific Softw. Eng. Conf. Can*, pp. 151–158, 2015.

[42] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," *IEEE Int. Work. Conf. Min. Softw. Repos.*, pp. 247–256, 2013.

[43] K. Koochekian Sabor, A. Hamou-lhadj, and A. Larsson, "DURFEX : A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports," pp. 240–250, 2017.

[44] P. K. Novak, N. Lavrač, and G. I. Webb, "Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining," *J. Mach. Learn. Res.*, vol. 10, no. Feb, pp. 377–403, 2009.

Appendix A:

Figure A1.
MAP values for Firefox and Gnome for various ranks

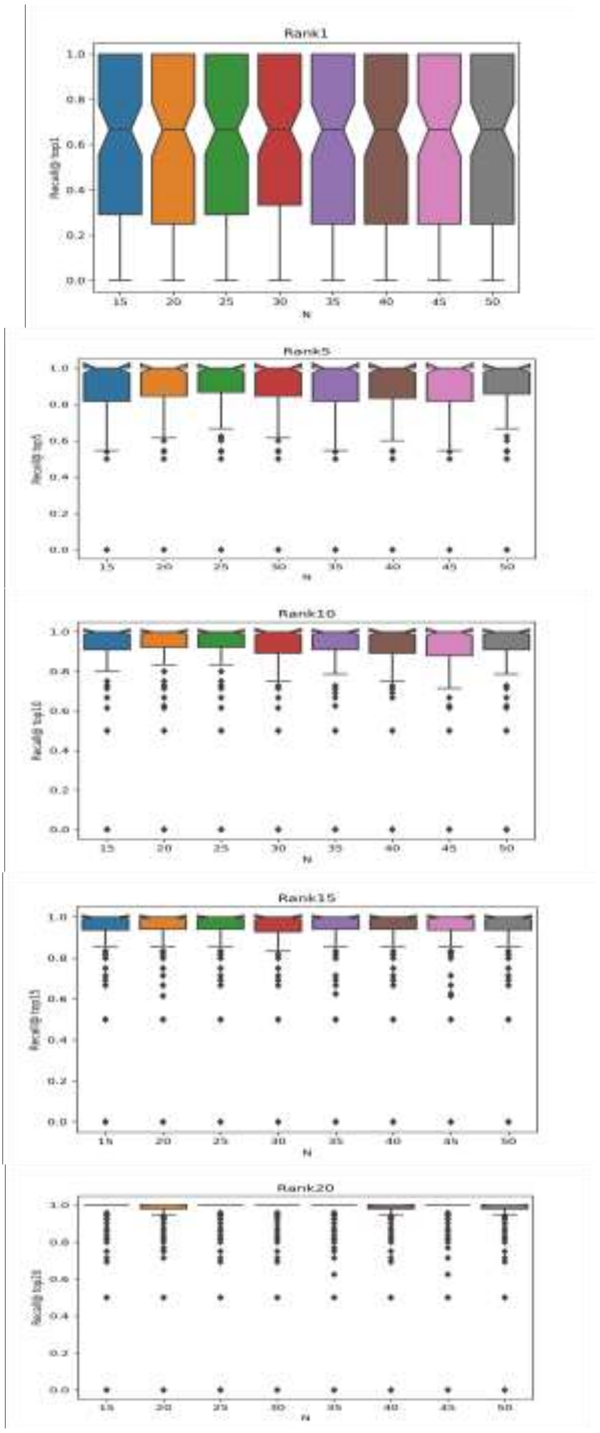

Fig. A1. Recall@rank-k for Firefox dataset.

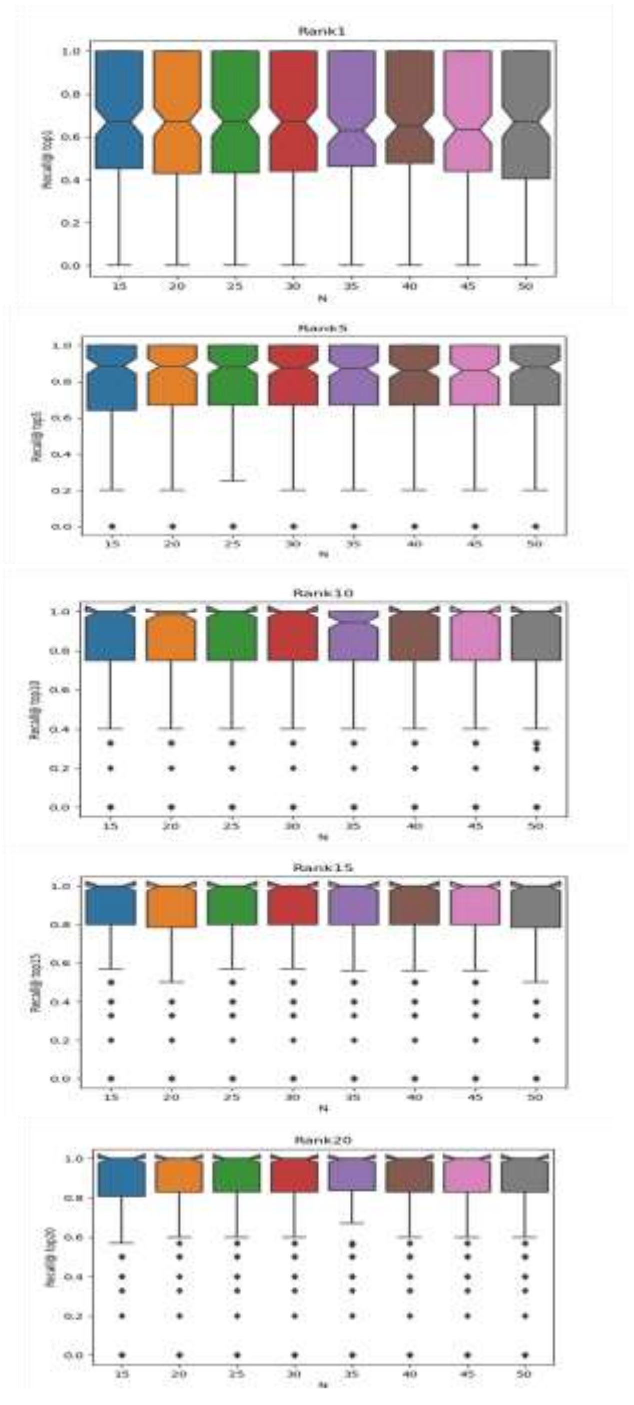

Fig. A2. Recall@rank-k for GNOME dataset.

Appendix B: Results of recall rate@rank-k with different HMM state numbers

Table B1
Median of recall rate@rank-k with different HMM state numbers using Firefox dataset.

Rank	**Number of Hidden States**							
	15	**20**	**25**	**30**	**35**	**40**	**45**	**50**
1	59.08%	59.70%	59.26%	59.81%	57.91%	58.21%	58.86%	58.50%
2	74.28%	76.36%	76.01%	75.76%	74.32%	75.23%	76.79%	75.67%
3	80.09%	81.20%	81.90%	81.84%	80.80%	80.55%	82.10%	82.95%
4	84.64%	85.44%	86.19%	85.08%	85.29%	85.24%	85.36%	85.19%
5	86.92%	86.55%	87.46%	87.33%	86.41%	87.05%	86.58%	87.11%
6	87.95%	88.53%	88.14%	88.17%	88.16%	88.10%	87.44%	88.52%
7	88.62%	89.03%	88.69%	88.64%	88.67%	88.71%	87.84%	88.65%
8	88.91%	89.48%	88.91%	88.80%	88.87%	88.88%	88.62%	88.82%
9	89.15%	89.57%	89.31%	89.20%	89.08%	89.19%	88.83%	89.03%
10	89.88%	89.90%	89.76%	89.52%	89.53%	89.54%	89.22%	89.47%
11	90.10%	90.00%	89.95%	89.84%	89.73%	89.76%	89.37%	89.57%
12	90.25%	90.12%	89.99%	89.96%	89.87%	89.85%	89.46%	89.86%
13	90.25%	90.12%	90.15%	90.04%	89.87%	89.94%	89.79%	90.04%
14	91.29%	90.81%	90.64%	90.58%	90.57%	90.72%	90.28%	90.55%
15	91.37%	91.36%	91.19%	90.65%	90.75%	91.28%	90.88%	91.22%
16	91.52%	91.48%	91.28%	90.81%	91.05%	91.36%	91.04%	91.31%
17	91.98%	91.80%	91.74%	91.66%	91.37%	91.92%	91.42%	91.77%
18	92.02%	92.26%	91.77%	91.77%	91.41%	91.98%	91.46%	91.77%
19	92.02%	92.26%	91.77%	91.84%	91.90%	91.98%	91.79%	91.77%
20	92.08%	92.33%	92.08%	91.84%	92.04%	92.02%	91.79%	91.77%
MAP	75.77%	76.44%	76.29%	76.40%	76.15%	76.33%	76.29%	76.26%

Table B2
Median of recall rate@rank-k with different HMM state numbers using GNOME dataset.

Rank	Number of Hidden States							
	15	**20**	**25**	**30**	**35**	**40**	**45**	**50**
1	62.99%	62.74%	62.66%	63.39%	63.15%	63.72%	62.18%	62.50%
2	73.21%	73.54%	73.54%	73.30%	73.13%	72.81%	73.38%	73.21%
3	76.95%	77.27%	76.54%	76.87%	76.70%	76.79%	76.54%	77.03%
4	78.57%	78.73%	78.17%	78.08%	78.49%	78.41%	78.33%	78.65%
5	79.79%	79.79%	79.38%	79.79%	79.71%	79.38%	79.30%	80.19%
6	80.76%	81.25%	80.60%	81.09%	81.33%	80.68%	80.52%	81.09%
7	82.06%	82.47%	81.74%	82.39%	81.98%	81.98%	81.57%	81.98%
8	83.36%	83.36%	82.47%	83.20%	82.79%	82.95%	82.71%	82.87%
9	83.85%	83.93%	83.36%	84.17%	83.44%	83.77%	83.52%	83.60%
10	84.50%	84.50%	84.09%	84.74%	84.25%	84.17%	84.58%	84.82%
11	85.23%	85.23%	84.90%	85.39%	85.15%	84.98%	85.06%	85.39%
12	85.63%	86.36%	85.63%	85.88%	85.88%	85.96%	85.80%	85.96%
13	86.53%	87.09%	86.28%	86.61%	87.18%	86.61%	86.53%	86.85%
14	87.18%	87.58%	87.01%	87.34%	87.74%	87.01%	87.26%	87.26%
15	87.91%	87.91%	88.07%	88.07%	88.23%	87.99%	88.31%	87.91%
16	88.56%	88.56%	88.88%	88.56%	88.72%	88.47%	88.80%	88.72%
17	89.04%	88.96%	89.69%	88.96%	89.12%	89.29%	89.20%	89.37%
18	89.53%	89.53%	90.26%	89.37%	89.53%	89.61%	89.77%	89.69%
19	90.02%	89.85%	90.58%	90.02%	90.34%	90.26%	90.26%	89.94%
20	90.26%	90.42%	90.99%	90.58%	90.91%	90.99%	90.58%	90.34%
MAP	71.32%	71.33%	71.13%	71.53%	71.37%	71.57%	70.86%	71.11%