An Effective Method for Detecting Duplicate Crash Reports Using Crash Traces and Hidden Markov Models

Neda Ebrahimi K.
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, QC, Canada
n_ebr@ece.concordia.ca

Md. Shariful Islam
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, QC, Canada
mdsha_i@ece.concordia.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, QC, Canada
abdelw@ece.concordia.ca

Mohammad Hamdaqa
School of Computer Science
Reykjavik University
Reykjavik, Iceland
mhamdaqa@ru.is

ABSTRACT

When a software system crashes, crash information from user's machine is sent to the developers of the system for repair. For software systems with a large client base (such as Eclipse, Web browsers, etc.), the number of reports that are submitted every day can be quite high. Managing these reports is known to be a tedious and a time consuming task. Fortunately, not all crashes are caused by new faults. Studies have shown that around 30% of the reported crashes are duplicates of previously reported ones. Automatic detection of duplicate crash reports can then reduce the time and effort dealing with crash reports. In this paper, we introduce a novel method for detecting duplicate crash reports using crash traces and Hidden Markov Models. We show that our approach outperforms existing methods in detecting duplicate crash reports.

Keywords

Mining Bug Repositories Duplicate Crash Reports, Hidden Markov Model, Software Maintenance.

1. INTRODUCTION

Bug reports carry important information that can be used by developers to detect and fix bugs [26, 27]. To manage the reporting of bugs, many software projects rely on bug tracking systems (BTSs) such as Bugzilla [10], WER (Windows Error Reporting) [20], and Apple Crash Reporter [5]. Bug reports can be collected automatically using crash reporter or submitted manually by users and developers.

However, due to the large number of bug reports submitted every day, the bug handling process tends to be challenging. Studies show that a significant number of submitted bug reports are duplicates of already reported defects. For example, a study by Lazar et al. on Eclipse, OpenOffice, Mozilla, and Netbeans shows that up to 23% of the total reports are duplicates [21]. A bug report is considered a duplicate if it refers to either the same previously reported abnormal execution or a distinct behavior of an existing bug report [25, 33].

CASCON'16, Oct-Nov 2016, Toronto, Ontario, Canada.

When a bug report is submitted, a triager[1] must detect whether a bug report is a duplicate of an existing one by inspecting similar reports and functions [42]. Moreover, to increase the productivity of triagers, there is a need for techniques that can automatically detected duplicate bug reports [4, 16, 22, 23, 25, 33, 40].

Existing techniques for detecting duplicate reports can be divided into two categories. The first category encompasses techniques that rely solely on the description of the reports [1, 16, 28, 38–40]. The problem with the description is that it is expressed in natural language and as such it tends to be informal and hence not quite reliable. To address this, researchers have turned to more formal crash report data such as stack traces [11, 12, 19, 43]. These techniques model information in historical stack traces and use the resulting model to classify incoming reports.

In our previous work, we presented CrashAutomata [13]. CrashAutomata builds a model from historical crash reports (more precisely their stack traces) that is used to classify an incoming report. The model is based on varied-length n-grams and automata. Unlike existing techniques, CrashAutomata takes advantage of the generalization aspect of automata, making it possible to build a representative model of crash reports, reducing the number of false positives. We showed that CrashAutomata outperforms the state of the art duplicate bug report detection techniques such as CrashGraph [19].

In this paper, we further improve CrashAutomata by leveraging the use of Hidden Markov Models (HMM). HMM is a machine learning technique that is used in speech recognition, DNA sequence analysis [14], and language processing [29, 31]. HMM is especially designed for sequential data analysis [6, 8]. We use HMM to model function calls of stack traces of historical bug reports to build a training model that is later used to classify incoming bug reports.

The remaining parts of this paper are organized as follows: In Section 2, we present primitive information on bug reports in Bugzilla bug reporting system as well as stack traces in Mozilla.

[1] A triager is person (i.e., tester, developer or project manager) who is responsible for prioritizing bug reports, identifying duplicate reports [3, 43] and assigning reports to an appropriate developers.

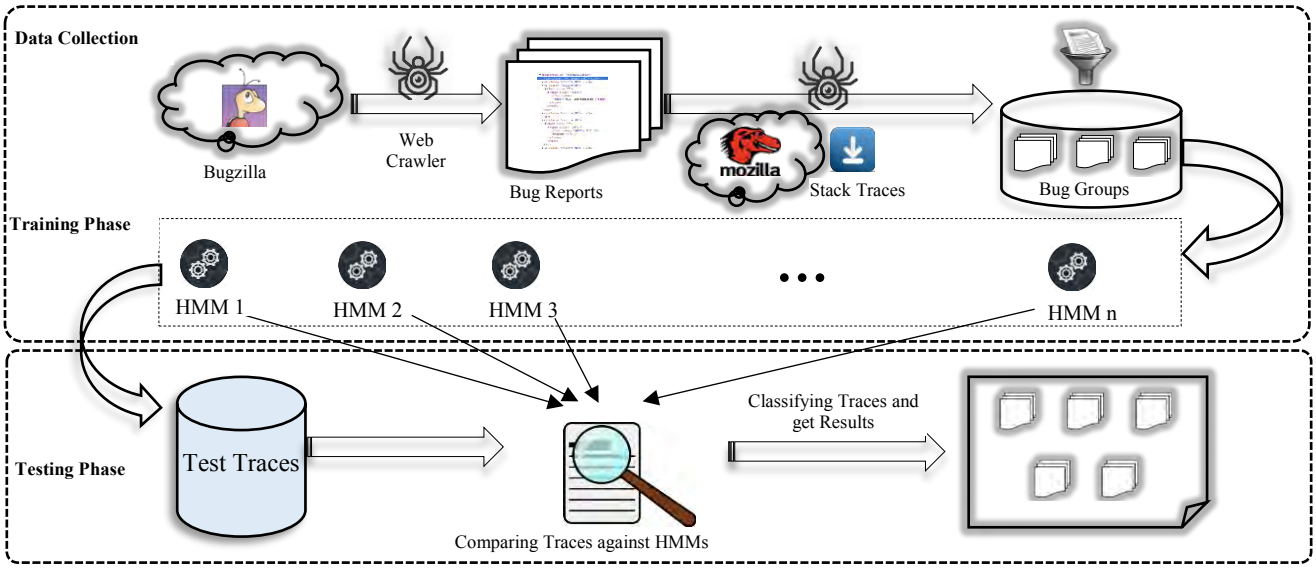

Figure 1. An overview of proposed approach

In Section 3, we explain the proposed approach. In Section 4, we evaluate the effectiveness of our work when we applied HMM on Bugzilla Mozilla bug reports, then we compare the HMM approach to CrashAutomata [13]. Section 5 is discussion about the work followed by threats to validity and related work in sections 6 and 7 respectively. Finally, we conclude the paper and sketch future directions in Section 8.

2. PRELIMINARIES

Mozilla bug repository contains crash reports submitted for many systems, such as Firefox, Thunderbird, SeaMonkey, Bugzilla, and Firefox Android. A crash report includes information that is collected during a crash from the user's machines such as the OS, the product version, the crash date and time, the stack traces, and other information.

In this paper, we use stack traces to detect duplicate reports. A stack trace can be defined as a sequence of function calls, called frames, $F=f_0, f_1, \ldots, f_n$, where f_0 represents the last function that is executed before the crash (also called the top function signature of the stack) and f_n is the last function in the stack trace. Table 1 shows an example of part of a stack trace of one of the crash reports in Mozilla. The stack trace in this example contains four frames starting from Frame 0. Each frame includes information about the running functions such as the module name, the function signature, and the source file where the function is defined. In this example, the program crashed at Frame 0 (mozalloc_abort(char const* const)).

When a bug report is submitted, a triager starts by first detecting if the received bug report is a new or a duplicate report of a previously reported one. If the bug report is considered new, the triager will assign the bug report to a developer to provide a fix. Bugzilla holds information about the reported bugs, as well as the tracking changes made to a bug by developers. If a bug report is a duplicate, a link is created to the master bug report.

Since stack traces keep a sequence of the running functions when a crash happens, crash reports are normally categorized according to the top function of their stack traces into groups called 'buckets' or crash types. Crash reporting systems follow different algorithms in grouping their crash reports. For example, WER (Windows Error Reporting) uses more than 500 proprietary heuristics for grouping its crash reports into buckets for Microsoft products [6, 10]. However, in Mozilla, crashes are only grouped according to their top method signature (buckets). As a result, various number of crash reports may be assigned to unrelated crash groups. Moreover, many related crashes may be distributed over wrong buckets. This makes it harder for a developer to recognize the duplication.

In this paper, we introduce a novel technique to automatically detect duplicate bug reports, by leveraging stack traces inside bug reports and their related duplicates. Our proposed approach allows developers to detect duplicate bug reports with high accuracy; consequently, it lowers bug-fixing time.

Table 1. An example of a stack trace in Mozilla

Frame	Module	Signature	Source
0	mozglue.dll	mozalloc_abort(char const* const)	memory/mozalloc/mozalloc_abort.cpp:33
1	xul.dll	NS_DebugBreak	xpcom/base/nsDebugImpl.cpp:403
2	xul.dll	NS_ProcessNextEvent(nsIThread*, bool)	xpcom/glue/nsThreadUtils.cpp:290
3	xul.dll	nsThread::ProcessNextEvent(bool, bool*)	xpcom/threads/nsThread.cpp:1029

3. THE APPROACH

Figure 1 shows an overview of the proposed approach. The approach consists of two phases: The goal of the first phase is to generate a Hidden Markov Model (HMM) for each bug group. In this phase, we start by collecting stack traces and creating bug groups. Then, we use the collected stack traces to train an HMM for every group. The goal of the second phase is to use the generated HMM to classify incoming bug

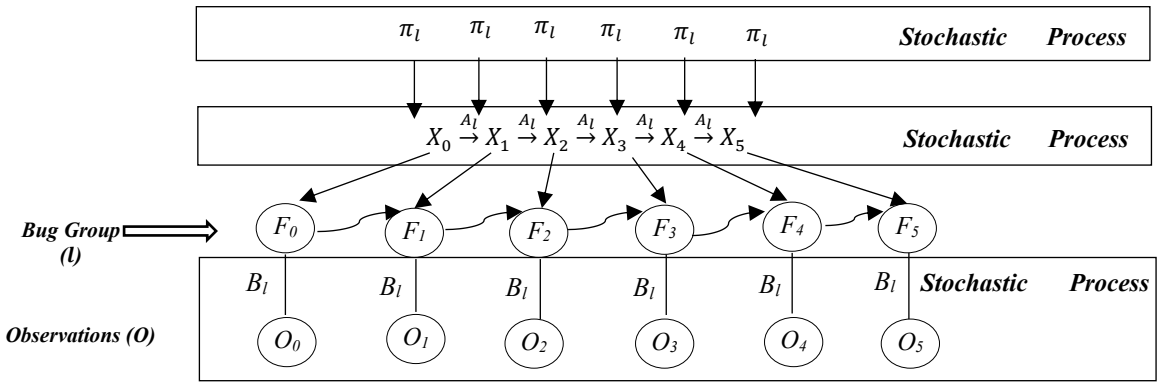

Figure 2. An HMM Model (λ_l) for a Bug Group (l)

reports by comparing their stack traces against the trained HMMs. We discuss each phase in details in the following subsections.

3.1 Generating an HMM for each bug group

3.1.1 Creating Bug Report Groups

In order to collect stack traces of duplicate bug reports, we implemented a web crawler to download bug reports in Bugzilla. We parsed the bug reports to extract duplicate bug reports as well as their corresponding stack traces in Mozilla. We divided the bug reports into groups where each group contains a master bug report and its duplicates. We will use these groups in the case study to derive training and testing sets so as to assess the effectiveness of our approach. Bug reports that do not have duplicates are excluded from the dataset. The details of the dataset used in this study are presented in Section 4.1.

3.1.2 Generating HMM

An HMM is a very effective statistical approach to model the behavior of system over time [31]. The behavior of a system can statistically be represented as a row of stochastic processes where the probability distribution sums to one.

An HMM uses three row stochastic processes. The first is the states and transition probability distribution (A) of a system in a Markov process. The second is the observation probability distribution (B) of observation sequences that comes from the temporal order of the stack traces. Finally, the third is the initial state probability distribution (π) of each hidden state in the Markov process. The first row is usually hidden in an HMM. The only event is the second one, the observation sequence (B) that is associated with the hidden states of the Markov process. An HMM can precisely learn the three stochastic processes A, B, π in a standard manner. Figure 3 illustrates a generic topology of an HMM, $\lambda = (A, B, \pi)$, the notations in the figure are adopted from [36].

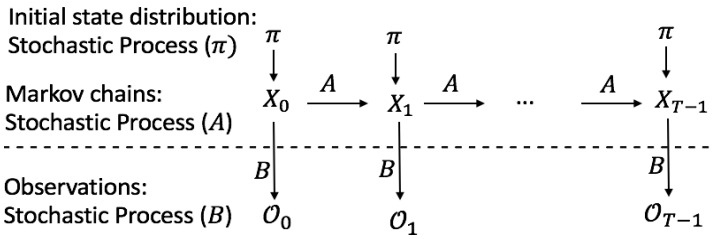

Figure 3. A generic topology of an HMM

Number of Hidden States (N): To learn an HMM model, we have to set the number of hidden states (N) in Markov process. Let the distinct states in a Markov process be S_i, $i = \{0,1,\dots,N-1\}$ and the notation $X_t = S_i$ represent the hidden state sequence S_i at time t.

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

State Transition Distribution (A): 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 denoted as:

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

Here, the transition from one state to the next is a Markov process of order one [31]. The order one means the next state depends only on the current state and its probability value. As the original states are "hidden" in HMM, we cannot directly compute the probability values in the past. But we are able to observe the observation symbols for the current state S_i at time t from the given observation sequence $\mathcal{O}$ to learn an HMM model.

Observation Symbol Distribution (B): The second row stochastic process is the observation symbol probability distribution matrix $B = \{b_j(R_k)\}$. B is an $N \times M$ dimensional matrix that 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 denoted as:

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

Initial State Distribution (π): 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 denoted as:

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

3.1.3 Training the HMM

The behavior of a system can be discrete (e.g., symbols from a finite alphabet) or continuous (e.g., signals from a speech, music, etc.). In our case, the behavior of each bug report can be represented as a discrete sequence of function calls. Figure 2 shows a general view of an HMM model where a sequence of function calls (F_i) of a bug report group is mapped to an observation sequence (O). Since a discrete HMM is a stochastic process for sequential data [31, 36], we used the observation sequence (O) to learn the behavior of each bug report group, .Thus, a well-trained HMM model using all the discrete sequences for a bug report can be used as a potential signature for detecting the duplicate bug report.

Practically, training a well-fit HMM using a discrete sequence of observation $\mathcal{O}$-$(\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$, means maximizing the likelihood, i.e., maximizing the probability of $P(\mathcal{O}|\ \lambda)$ on the parameters space represented by A, B, and π. The Baum-Welch (BW) algorithm is one of the most commonly used algorithms for learning the HMM parameters [37]. The most remarkable aspects of the BW algorithm is the Forward-Backward (FB) algorithm [36] because FB follows an iterative procedure to estimate the HMM parameters. In each iteration, FB can efficiently re-estimate the HMM model itself, if $P(\mathcal{O}|\ \lambda)$ increases. We can set a predetermined threshold and/or set a maximum number of iterations to stop it. We have also chosen the BW algorithm to train an HMM model for a bug group by setting the maximum number of iterations to 25. This number has been chosen though experimentation as we noticed that varying the number of iterations did not show any significant effect on the performance. This said, we set the value of the number of iterations to 25 to avoid overfitting and to reduce the learning time [37].

To train an HMM, we have to set the number of distinct symbols (M) for a fixed number of states (N). In our case, to select the value of M, we count all the distinct functions over all the discrete sequences for all bug report groups. For the Bugzilla dataset, we have M = 4895 over 3651 discrete sequences for 90 bug groups. To select the optimum number of hidden states (N), we have trained different discrete-time ergodic HMMs with various N values 5, 10, 15,…, 50 while always keeping M fixed. We obtained best accuracy for N = 25 (see Section 4.2 for more details). We build an HMM model for each bug report group by keeping M = 4895 and N = 25 fixed. At the end, the HMM that gives the highest log-likelihood value was selected as a model for that bug report group.

3.2 Classification

Let there are L number of unique bug report groups (l) in Mozila bug dataset, where $l = \{0,1, \ldots, L\}$. Therefore, we have $\lambda_l = (A_l, B_l, \pi_l)$ distinct trained HMM models for the fixed values of M = 4895 and N = 25. Let the Bugzilla bug reporting system generates a new bug report (i.e., the stack traces in Mozilla) and the corresponding sequence of observation is $\mathcal{O} - (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1})$. We have to compute the log-likelihood or the probability $P_l(\mathcal{O}|\gamma_l)$ for a trained HMM model (λ_l) as a score of this new observation sequence $(\mathcal{O})$. We use the FB algorithm to compute the scores or $P_l(\mathcal{O}|\gamma_l)$. As we have L trained HMM models (λ_l), we get L scores for a new observation sequence $\mathcal{O}$. If the new observation sequence $\mathcal{O}$ (i.e., the new bug report) is duplicate (let say the duplicate bug report group is l'), the score of that trained HMM model $(\lambda_{l'})$ should be significantly higher. That is, the probability of $P_{l'}(\mathcal{O}|\gamma_{l'})$ should be maximized as:

$$P_{l'}(\mathcal{O}|\gamma_{l'}) = \max_{P_l}[P_l(\mathcal{O}|\gamma_l)], \qquad l = \{0,1, \ldots, L\}$$

As described in the previous section, we construct an HMM for every single bug group in our dataset to classify the stack traces of incoming bug reports. We change the sequences of an incoming stack traces with the function IDs we have assigned in our training set and if a function does not exist in the ID list, it will be assigned a new function ID. Then, the incoming trace is compared to the HMM of each bug report group. A bug report is considered as duplicate of bug reports of the bug group whose HMM has the highest log-likelihood.

We measure the performance of our HMM method by first using a confusion matrix and then computing precision, recall and F-measure for all the bug groups and considering different state numbers of N=5, 10, 15,…, 50 when building HMMs (Table 3). The confusion matrix stores correct and incorrect predictions made by a classifier [44]. For example, if traces belonging to a bug report group are classified by HMM as the same bug group, they are True Positives (TP). Contrarily, if there exist traces in a bug group in which HMMs classified them in other bug groups rather than their real bug group, they are considered as False Negatives (FN). Finally, if traces of other bug groups are classified to be related to a bug group, these are treated as False Positives (FP). For a bug group G_i, we measure TP, FP, and FN as follows:

- TP_{Gi} = The number of traces that are correctly classified
- FP_{Gi} = The number of traces of the other buckets that are classified as bucket G_i
- FN_{Gi} = The number of traces of bucket G_i that were classified as belonging to other buckets other than G_i

We derive precision and recall for each bug group, G_i, as follows. Note that a high FP will reduce precision, whereas a high FN will reduce recall:

$$Precision\ (G_i) = \frac{TP_{Gi}}{TP_{Gi} + FP_{Gi}}$$

$$Recall\ (G_i) = \frac{TP_{Gi}}{TP_{Gi} + FN_{Gi}}$$

$$F_{measure}(G_i) = 2 * \frac{Precision(G_i) * Recall(G_i)}{Precision(G_i) + Recall(G_i)}$$

4. EVALUATION

In this section, we evaluate our approach by applying it to Bugzilla Mozilla bug reports. We measure the accuracy of our method using the precision, recall and F-measure. Additionally,

we examine the HMM with different number of states (N) to achieve the highest accuracy. We also, compare the proposed approach with our previous method (CrashAutomata [13]), as both use stack traces of Bugzilla Mozilla to detect duplicates.

Table 2. Properties of the dataset.

# of bug report groups	*Total # of bug report duplicates*	*Total # of crash traces*
90	210	3,651

Table 3. Results of varying number of states in our HMM

Number of States	TPR(%)	FPR(%)	FNR(%)	Precision	Recall	F-measure
5	94.0	3.3	6.0	96.9	94.0	94.9
10	94.5	2.9	5.5	97.3	94.5	95.5
15	94.5	2.9	5.5	97.3	94.5	95.5
20	94.5	2.8	5.5	97.4	94.5	95.5
25	94.6	2.8	5.4	97.4	94.6	95.6
30	94.4	2.8	5.6	97.3	94.4	95.4
35	94.5	2.8	5.5	97.4	94.5	95.5
40	94.3	2.6	5.7	97.4	94.3	95.5
45	94.5	2.8	5.5	97.4	94.5	95.5
50	94.3	2.7	5.7	97.5	94.3	95.5

4.1 Dataset: Bugzilla Mozilla

We chose Bugzilla Mozilla as our dataset because this is a well-known open source bug repository used by many researchers. Additionally, unlike other bug repositories (e.g., Eclipse and Gnome) stack traces are not located in the description part of bug reports. Instead, if there exist any related crash reports in Mozilla, a link to the crash report is provided. Meaning, one does not need to parse the description part to extract stack traces. However, for Mozilla only crash reports reported for the period of one year are available to the public.

In order to collect stack traces of duplicate bug reports, we implemented a web crawler to download the entire reported bugs in Bugzilla Mozilla up to January 2016 for the bug report IDs from 65000 to 370000 covering all existing products in Bugzilla Mozilla. For each crash report, we randomly downloaded 250 stack traces (if exist), and then a parser collected bug reports to extract duplicate bug reports as well as their linked stack traces in Mozilla. Bug reports are categorized in new groups according to their master bug reports. Therefore, each bug report group holds stack traces of master bug report beside the stack traces of its duplicates. We filtered bug report groups to keep only bug report groups with one master bug report and at least one duplicate. In total, our dataset consists of 3,651 stack traces, grouped into 90 distinct bug report groups labelling from 1 to 90 with an overall number of 210 duplicates (see Table 2).

4.2 Results

In our dataset, we selected the typical 70% of the stack traces for training purpose while we kept 30% for the testing set. Out of our 90 groups, only 73 bug report groups had traces in their test set because of a lack of number of traces.

We built an HMM for each bug report group by varying the number of states (N = 5, 10, 15,…, 50) in order to determine the state number that achieves best classification results. We also calculated precision, recall and F-measure for every bug report group. Table 3 shows the results of our method with different state numbers. Although, the results are very close, we achieved best results by having N = 25.

Figure 4 compares the evaluation metrics (Precision, Recall and F-measure) for the achieved result of HMM with N = 25. We obtained high median of 92 for the F-measure. For generating boxplots, we only kept the unique values to show the range of the achieved results and the volume of each range if not shown here (you may find the percentage of each range in Figures 6 to 8).

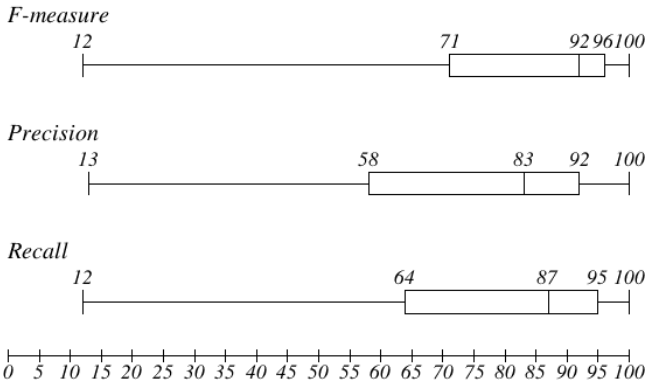

Figure 4. Precision, Recall and F-measure Boxplot applying proposed HMM

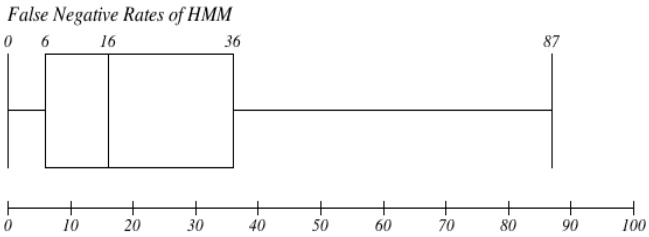

Figure 5. False Negative rates for HMM in all bug report groups.

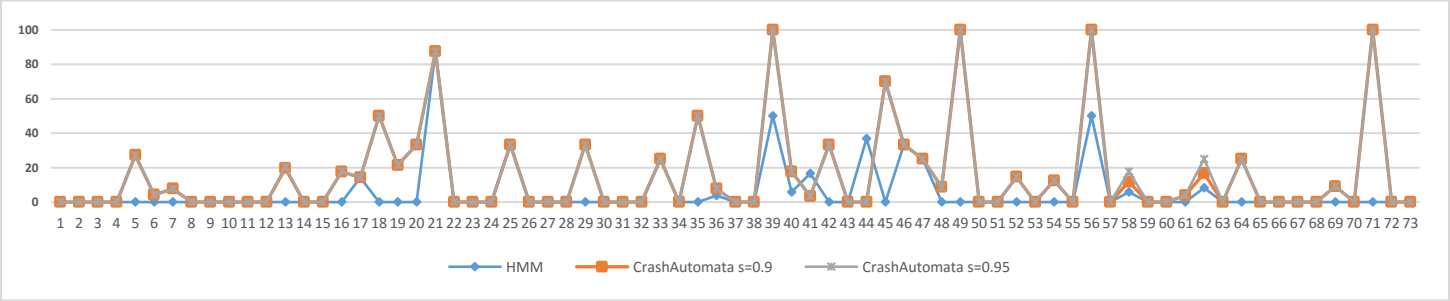

Figure 6. Comparing False Negative rates.

The false negative rate of all bug report groups is shown in a boxplot in Figure 5. The main reason of the high number of false negatives in some groups is the low number of existing samples in training and testing. Among 90 bug report groups only 73 bug groups have more than three stack traces in their bug groups. However, the median of 16 is an acceptable rate for groups with a high number of traces.

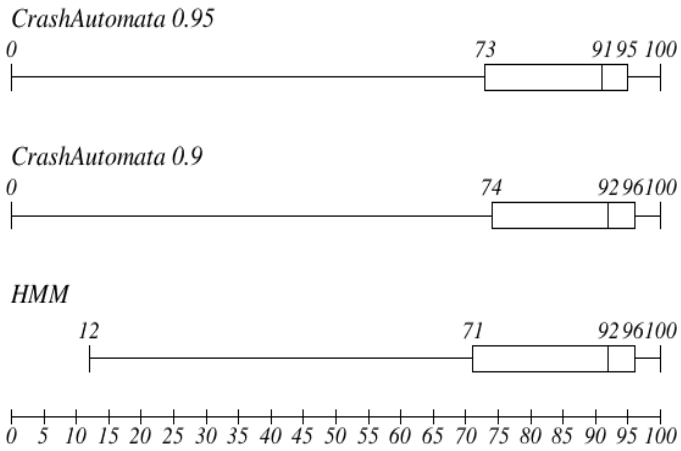

Figure 7. Comparing F-measure for HMM and CrashAutomata

4.3 Comparison with CrashAutomata

In this section, we compare the proposed method with our earlier work, CrashAutomata [13]. We implemented CrashAutomata to detect duplicate crash reports of Mozilla. Crash Automata aggregates the presentation of each bug report group by constructing a generalized automaton where the states are varied-length N-grams resulted from the most frequent N-grams considering a threshold α. The threshold α=0.9 produced the best accuracy for CrashAutomata. Also, we showed that CrashAutomata outperforms CrashGraph [19] by achieving higher F-measure and reducing the number of misclassified traces.

To compare CrashAutomata with the proposed approach, we run CrashAutomata using the same dataset as the ones described in Section 4.1. Since with CrashAutomata, we obtained best results with a threshold α=0.9, in this study also we only take this threshold into consideration. We evaluate the results with two similarity metrics τ=0.9 and τ=0.95.

We compute precision, recall and F-measure for both methods. The results are shown in Figure 9. As the figure shows, the results are extremely close. A noticeable difference is the minimum of F-measures that is zero for CrashAutomata but, at least 12% for HMM in all bug groups. One reason may be the logic behind the HMM classifier when it builds the HMM with

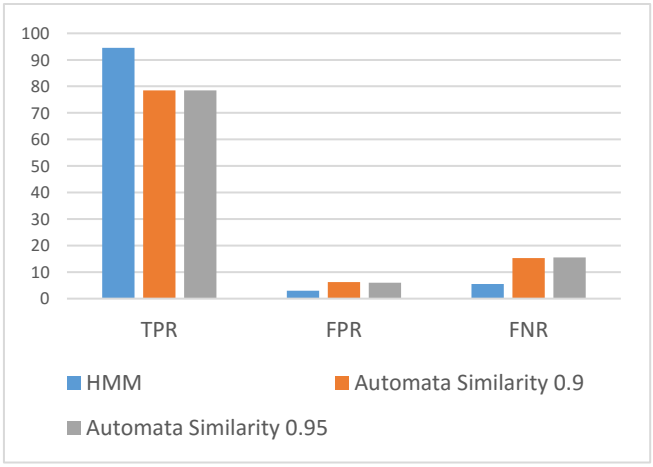

Figure 8. True Positive, False Positive and False Negative rates for HMM and CrashAutomta.

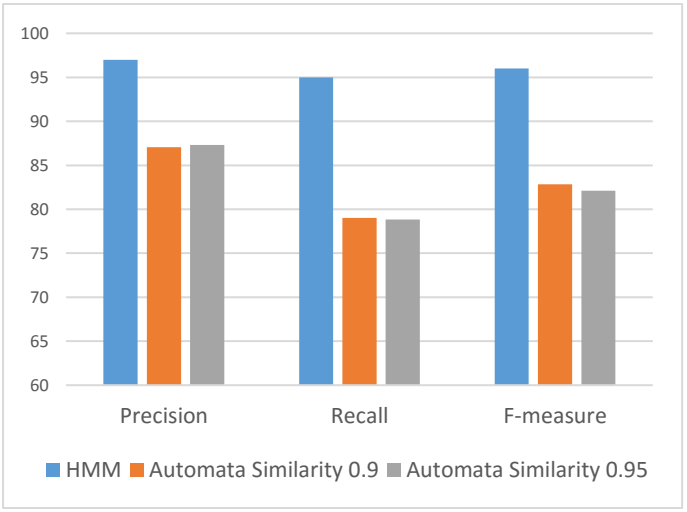

Figure 9. Comparing Precision, Recall and F-measure for HMM, CrashAutomta with similarity 0.9 and 0.95.

a maximum likelihood. In this case, every trace will be assigned to at least one group.

Figure 6 compares the details of False Negative rates of HMM and CrashAutomata (with similarity of 0.9 and 0.95). It is notable that in almost all bug groups HMM resulted in less False Negative rates than the CrashAutomata. For example, in bug groups 49 and 71, HMM was able to successfully assign traces to the correct bug group where CrashAutomata falsely assigned 100% of the traces to other bug groups. The False Negative rate difference of two approaches for bug groups 18, 35, 45 and 56 shows significant distinction of 50% in average where this difference for bug groups 5, 13, 25, 29, 33, 42, 52 and 54 is between 20% and 40%.

In general, HMM produces less False Negatives than the CrashAutomata. Only in couple bug groups (bug groups 41 and 44), HMM shows higher False Negative rates than the CrashAutomata.

Additionally, the highest False Negative rate belongs to bug group 21 (84%) that is also similar to the CrashAutomata. When the False Negative rate for 4 bug groups is 100% in CrashAutomata, by applying HMM we have no bug group with 100% False Negative rate.

5. Discussion

Although increasing the number of states may improve the accuracy of the HMM, as it is shown in Table 3, in our case N = 25 results in the best accuracy. However, a change in the number of states may not have a significant impact on the accuracy. Also, the False Negative rate range for our dataset varies between 0 and 87, which demonstrates better results than the CrashAutomata. Having no bug group resulting in 100% False Negative rate is also another notable point for HMM. HMM is able to detect duplicate bug reports with high accuracy even with the lower number of traces.

Additionally, using an HMM-based approach, we obtained approximately 10% higher true positive rate than CrashAutomata. Since HMM classifies every single bug report into existing bug report groups, we have less false negatives and consequently higher F-measure for HMM comparing to CrashAutomata.

We only observed the results for rank one (the top likelihood we achieved) of the HMM and we believe that delivering more ranks to the developers may further enhance the results.

As described before, for the boxplots we only show the distribution of values and not the volume of each values. We show the details of the obtained results and compare it to the CrashAutomata with two similarities in Figure 6.

Finally, our results imply that the stack traces alone carry crucial information inside crash reports that by only using stack traces of a bug report a developer can detect duplicate bug reports with a very high accuracy. Moreover, using stack traces may eliminate the effort by users in reading natural languages in summary and description parts of bug reports which lead to more time and effort saving.

6. THREATS TO VALIDITY

In this section we discuss threats to validity that may invalidate the results presented in this paper.

Number of states (N) and observation sample: As we used HMM for our classifier, one of the vulnerable variables and a threat to validity in implementing HMM is the number of states (N) and the number of distinct symbols (M). To address this issue, we examined the results by varying N and showed that in our dataset N = 25 yields best results. We also observed that increasing N may result in an increase in the running time without necessarily improving the accuracy of the approach.

Moreover, we know that a well-fit HMM model highly depends on how we define our observation samples [31]. In our system, the length (T) of the training sequences is variable and the number of observation sequences is also variable for each trained HMM model. As the number of observation samples is limited for a bug report group in our dataset, we cannot evenly divide it for every bug report group. We also cannot evenly divide the length of observation sequence (T), because the order of executions of different bug report groups is also different. In fact, the discriminations in the length of observation sequences (T) is also effective in classification as we learned an HMM model separately for each bug report group.

The selected crash reports dataset: The selected dataset is an external threat to validity. In our experiment, we selected bug reports from Bugzilla Mozilla. While Bugzilla is one of the most commonly used open source bug repositories for researchers, we intend to use other datasets to generalize the results.

Number of stack traces: We downloaded 250 random stack traces of each crash report from Mozilla. Perhaps, if we choose a different number of stack traces, this will impact the results. We followed the common practice that was used by other researchers. Many researches select randomly 100 to 200 stack traces in their experiments. Besides, we believe that because of the high duplicate rate of stack traces in some crash reports and the lack of stack traces for many others (many crash reports have only one stack trace inside), our collected dataset is representative and valid for this study.

Ranking: In our method we did not apply ranking for getting results as we achieved high rate of precision, recall and F-measure. However, offering more ranks to the developers (e.g., rank of top 2, 5 and 10) may yield higher accuracy than only using the highest ranked label. On the other hand, increasing the number of suggested ranks may decrease performance..

Misclassified bug reports: HMM assigns all incoming bug reports to at least one of the existing bug groups. In the real world, if a new bug report is received by the system it should be assigned to a new bug group. To solve this issue, we may define a new bug group besides our existing groups to assign the unclassified bug groups to a new group instead of assigning it to incorrect bug group.

7. RELATED WORK

Duplicate detection of bug reports has been handled in various ways. Existing studies measure the similarity between the existing bug reports and incoming bug reports by either employing feature extraction and Natural Language Processing techniques and calculating the similarity of bug reports, such as Summary and Description [7, 16, 25, 39, 41–43], or adopting pattern matching on the execution information part of bug reports such as stack traces [12, 13, 16, 19, 35]. More focus is

dedicated to the similarity measurement of textual information of bug reports to overcome the unavailability of execution information in some bug repositories. Although, there exist studies that have been done in textual mining of bug reports [1, 2, 9, 17, 18, 24, 30], we only mention a few of the most recent ones as our main focus is using stack traces that accompany bug reports.

The work by Lin et al. [25] proposes a manifold correlation feature approach, which improves the previous SVM-based discriminative approach (SVM-54). This method combines the metrics calculated by CC-based scoring, TF-IDF-CFC weighting, BM25-based weighting and Word2vec weighting. Authors compare the results with SVM-54 scheme and proof that their SVM-SBCTC is able to improve the range of the performance up to 29% in top 5 recall rates. Another work by Tsuruda et al. [42] investigates the feasibility of deleting duplicate bug reports with unfinished bug reports by leveraging feature extraction and the number of words in a bug reports. Results observed that using words more than a certain number (e.g. about 100 and 80 words in Eclipse and OpenOffice, respectively) cannot improve the accuracy of detecting duplicate bug reports. The needed effort for identifying duplicate issues is in a study conducted by Rakha et al. [32]. They observed that more than 50% of duplicate reports are recognized within half a day [32]. The method builds a classification model to classify duplicates according to the needed effort and classifies duplicates with high precision and recall. This research also proofs the importance of duplicate detection for developers. The study by Lazar et al. [23] present an improved method for identifying textual similarity features. Selected similarity feature by TakeLab [34] are calculated by some binary classification method such as Naïve Bayes and Support Vector machines. Then, classifier detects bugs as supplicate or non-duplicates. This method shows an improvement of 3% to 6% over Alipour's method [2].

Wang et al. [43] introduced a method that uses text mining techniques to calculate the similarity of bug reports on both the textual parts (summary and description) and stack traces of bug reports. The suggested list is able to increase duplicate detections by around 25% in Firefox bug repository comparing to only using natural language information. CrashGraph [19], introduced by Kim et al. to identify duplicate crash reports in Microsoft's Windows Error Reporting (WER) system. A graph in representative of call stacks of a group (modules as nodes and calling sequences as edges) and every single incoming call stack is compared to only the representative graph of each group. CrashGraph demonstrates 71.5% and 62.4% precision and recall respectively. Our previous work, CrashAutomata [13] also uses the generalization attribute of automata for presenting call stacks of bug groups in Firefox. For each bug group an automaton is generated so that the incoming stack traces are examined against CrashAutomata of each group. If CrashAutomata accepts the crash stack, the crash report will be duplicate of the reports in that group.

8. CONCLUSION AND FUTURE WORK

In this paper, a new technique for detecting duplicate bug reports has been proposed in order to reduce the time and efforts required by triagers when working with bug reports.

We showed that using HMM and stack traces collected from historical crash reports it is possible to classify new incoming bug reports with high precision, recall and F-measure . For example, by applying the proposed approach on the set of stack traces collected from Bugzilla Mozilla, the F-measure value was (97%) in average.

We also successfully enhanced our previous work CrashAutomata [13] with HMM and compared the results of both approaches. The results indicate that when using HMM the number of false negatives and false positives can be reduced, since all traces are assigned to at least one bug report group.

In the future, we aim to use N-grams (as we use in our CrashAutomata) for generating the states in HMM. We anticipate that this should increase the overall accuracy of HMM. We also intend to implement our work in tools for analyzing trace information such as SEAT [15].

ACKNOWLEDGMENT

This study was partly supported by a grant from NSERC (the Canadian Natural Sciences and Engineering Council).

9. REFERENCES

[1] Aggarwal, K., Rutgers, T., Timbers, F., Hindle, A., Greiner, R. and Stroulia, E. 2015. Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge. (2015), 211–220.

[2] Alipour, A., Hindle, A. and Stroulia, E. 2013. A contextual approach towards more accurate duplicate bug report detection. *IEEE International Working Conference on Mining Software Repositories*. (2013), 183–192.

[3] Anvik, J., Hiew, L. and Murphy, G.C. 2005. Coping with an open bug repository. *2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse'05, October 16, 2005 - October 17, 2005*. (2005), 35–39.

[4] Anvik, J., Hiew, L. and Murphy, G.C. 2006. Who should fix this bug? *Proceeding of the 28th international conference on Software engineering - ICSE '06*. 2006, (2006), 361.

[5] Apple Crash Reporter: *https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AnalyzingCrashReports/AnalyzingCrashReports.html*.

[6] Bengio, Y. 1996. Markovian Models for Sequential Data. *Neural Computing Surveys*. 2, (1996), 129–162.

[7] Bettenburg, N., Premraj, R., Zimmermann, T. and Kim, S. 2008. Duplicate bug reports considered harmful... Really? *IEEE International Conference on Software Maintenance, ICSM*. (2008), 337–345.

[8] Bicego, M., Murino, V. and Figueiredo, M.A.T. 2003. Similarity-Based Clustering of Sequences Using Hidden Markov Models. *Machine Learning and Data Mining in Pattern Recognition*. (2003), 86–95.

[9] Brodie, M., Ma, S., Rachevsky, L. and Champlin, J. 2005. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*. 13, 2 (2005), 219–236.

[10] Bugzilla: *https://bugzilla.mozilla.org/*.

[11] Dang, Y., Wu, R., Zhang, H., Zhang, D. and Nobel, P. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. *Proceedings - International Conference on Software Engineering*. (2012), 1084–1093.

[12] Dhaliwal, T., Khomh, F. and Zou, Y. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. *IEEE International Conference on Software Maintenance, ICSM*. November 2009 (2011), 333–342.

[13] Ebrahimi Koopaei, N. and Hamou-Lhadj, A. 2015. CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata. *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering* (Riverton, NJ, USA, 2015), 201–210.

[14] Eddy, S.R. 1996. Hidden Markov models. *Current Opinion in Structural Biology*. 6, 3 (1996), 361–365.

[15] Hamou-Lhadj, A., Lethbridge, T.C. and Fu, L. 2005. SEAT: A Usable Trace Analysis Tool. *13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, {USA}* (2005), 157–160.

[16] Jalbert, N. and Weimer, W. 2008. Automated duplicate detection for bug tracking systems. *Proceedings of the International Conference on Dependable Systems and Networks*. (2008), 52–61.

[17] Jones, a. J. and Harrold, M.J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Automated Software Engineering*. (2005), 282–292.

[18] Jones, J. a., Harrold, M.J. and Stasko, J. 2002. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. (2002), 467–477.

[19] Kim, S., Zimmermann, T. and Nagappan, N. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. *Proceedings of the International Conference on Dependable Systems and Networks*. (2011), 486–493.

[20] Kinshumann, K., Glerum, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G. and Hunt, G. 2011. Debugging in the (very) large. *Communications of the ACM*. 54, 7 (2011), 111.

[21] Lazar, A., Ritchey, S. and Sharif, B. 2014. Generating duplicate bug datasets. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. (2014), 392–395.

[22] Lazar, A., Ritchey, S. and Sharif, B. 2014. Generating duplicate bug datasets. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. (2014), 392–395.

[23] Lazar, A., Ritchey, S. and Sharif, B. 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. (2014), 308–311.

[24] Liblit, B., Naik, M., Zheng, A.X., Aiken, A. and Jordan, M.I. 2005. Scalable statistical bug isolation. *ACM SIGPLAN Notices*. 40, 6 (2005), 15.

[25] Lin, M.J., Yang, C.Z., Lee, C.Y. and Chen, C.C. 2014. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software*. 0, (2014), 1–11.

[26] Lotufo, R., Malik, Z. and Czarnecki, K. 2015. Modelling the "hurried" bug report reading process to summarize bug reports. *Empirical Software Engineering*. 20, 2 (2015), 516–548.

[27] Nayrolles, M. and Hamou-Lhadj, A. 2016. BUMPER: A Tool for Coping with Natural Language Searches of Millions of Bugs and Fixes. *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016* (2016), 649–652.

[28] Nguyen, A.T., Nguyen, T.T.T.N., Lo, D. and Sun, C. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. (2012), 70.

[29] Onegin, E. 2014. Hidden Markov Models. Chapter 18 (2014), 1–21.

[30] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J.S.J. and Wang, B.W. Bin 2003. Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003. Proceedings.* 6, (2003), 465–475.

[31] Rabiner, L.R. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Ieee*.

[32] Rakha, M.S., Shang, W. and Hassan, A.E. 2015. Studying the needed effort for identifying duplicate issues. *Empirical Software Engineering*. (2015).

[33] Runeson, P., Alexandersson, M. and Nyholm, O. 2007. Detection of duplicate defect reports using natural language processing. *Proceedings - International Conference on Software Engineering*. (2007), 499–508.

[34] Šarić, F., Glavaš, G., Karan, M., Šnajder, J. and

Bašić, B.D. 2012. TakeLab: Systems for Measuring Semantic Text Similarity. *First Joint Conference on Lexical and Computational Semantics (*SEM)*. (2012), 441–448.

[35] Schröter, A., Bettenburg, N. and Premraj, R. 2010. Do stack traces help developers fix bugs? *Proceedings - International Conference on Software Engineering*. (2010), 118–121.

[36] Stamp, M. 2004. A revealing introduction to hidden Markov models. *Department of Computer Science San Jose State ...*. (2004), 1–20.

[37] Statistics, M. 2008. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains Author (s): Leonard E . Baum , Ted Petrie , George Soules , Norman Weiss Source : The Annals of Mathematical Statistics , Vol . 41 , No . 1 (Feb . *Statistics*. 41, 1 (2008), 164–171.

[38] Sun, C., Lo, D., Khoo, S.C. and Jiang, J. 2011. Towards more accurate retrieval of duplicate bug reports. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*. (2011), 253–262.

[39] Sun, C., Lo, D., Wang, X., Jiang, J. and Khoo, S.-C. 2010. A discriminative model approach for accurate duplicate bug report retrieval. *2010 ACM/IEEE 32nd International Conference on Software Engineering*. 1, (2010), 45–54.

[40] Sureka, A. and Jalote, P. 2010. Detecting duplicate bug report using character N-gram-based features. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. (2010), 366–374.

[41] Tian, Y., Sun, C. and Lo, D. 2012. Improved duplicate bug report identification. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. (2012), 385–390.

[42] Tsuruda, A. 2015. Can We Detect Bug Report Duplication with Unfinished Bug Reports ? (2015).

[43] Wang, X.W.X., Zhang, L.Z.L., Xie, T.X.T., Anvik, J. and Sun, J.S.J. 2008. An approach to detecting duplicate bug reports using natural language and execution information. *2008 ACM/IEEE 30th International Conference on Software Engineering*. (2008).

[44] Witten, I.H., Eibe, F. and Hall, M.A. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.