Analyzing and Predicting Concurrency Bugs in Open Source Systems

Paolo Ciancarini*†, Francesco Poggi*, Davide Rossi*† and Alberto Sillitti‡
*University of Bologna, Italy
{paolo.ciancarini, francesco.poggi5, daviderossi}@unibo.it
†Consorzio Interuniversitario Nazionale per l'Informatica, Italy
‡Innopolis University, Russian Federation
a.sillitti@innopolis.ru

Abstract*—*Background **Software systems are relying more and more on multi-core hardware requiring a parallel approach to address the problems and improve performances. Unfortunately, parallel development is error prone and many developers are not very experienced with this paradigm also because identifying, reproducing, and fixing bugs is often difficult.** ***Objective*** **The main goal of this paper is the identification of an approach able to: (i) identify solved concurrency-related bugs to characterize them and help retrospective activities; (ii) identify concurrency-related bugs as soon as they are entered in the bug management system to support bug triage phase and allocate them to more experienced developers.** ***Approach*** **To this end, the paper analyzes bugs related to concurrency looking at their specific characteristics using different machine learning methods to automatically distinguish them from other kinds of bugs based on the data available in the issue tracking systems and in the code repositories.** ***Results*** **The overall best models we developed for Apache HTTP Server and MariaDB have a precision of 0.985 and 0.814 and a recall of 0.876 and 0.629 when considering linked bugs (bug reports information in bug repository and the corresponding fix in the version control system) and a precision of 0.978 and 0.779 and a recall of 0.889 and 0.569 when considering only the information from bug reports.** ***Conclusions*** **Such results allow the development of an automated system able to classify such bugs and support developers in the bug triage process.**

***Index Terms*—concurrency; defects; prediction model; open source**

I. Introduction

To reduce the energy consumption of new devices and increase the battery life of the mobile ones, hardware manufacturers have changed deeply how they develop CPUs. In the last 10 years, to improve energy efficiency, the clock frequency of the CPUs has not increased significantly (even reduced in some cases) but the number of computational cores embedded in the CPUs has increased including both general purpose and special purpose ones (e.g., GPUs, DSPs, etc.). This change of architecture has a deep impact on software developers [7] [8]. In the single-core era, in most of the use cases, developers did not focus on performances while developing relying on the Moore's Law. Developers had just to wait some time and their software could run faster and faster on new CPUs without any modification. However, in the multi-core era, this is not true anymore. Software performances are not increasing without an explicit support of the multi-core architectures that require a completely different approach to software development.

Such different approach is not completely new since it derives from the parallel programming approaches. However, only a small percentage of developers were skilled in that since it was popular only in the fields where massive computation was needed (e.g., scientific computation, signal processing, computer graphics, etc.). Multi-core architectures have forced any kind of developers to deal with concurrent programming in almost any kind of software.

Concurrent programming is difficult since it is often affected by non-determinism due to the independent execution of the different threads and the related synchronization problems. Therefore, debugging this kind of software is often more difficult than single-thread code, also because detecting and replicating such defects is quite difficult.

Moreover, from a preliminary investigation focused on the Apache HTTP Server conducted by the authors [5] [6], we have found out that concurrency-related defects require the involvement of more developers and much longer discussions to get fixed compared to non-concurrency-related defects. For these reasons, characterizing concurrency-related defects and developing approaches to help developers during the bug triage phase to automatically identify the concurrency-related ones can improve how such defects are managed and the efficiency of the overall process.

The goals of the paper are the following:

- **G1:** Understanding and characterizing concurrency-related bugs to help developers in performing project retrospectives and improve the overall development approach.
- **G2:** Develop a classifier able to identify concurrency-related bugs when they are introduced in the issue tracking system. This classifier is intended to be used in the bug triage phase to help developers in assigning concurrency-related bugs to more experienced developers.

To this end, this paper analyzes two popular open source projects (Apache HTTP Server and MariaDB) that have been analyzed from many points of views [11] [30] but in this case we focus on the point of view of the defects related to concurrency analyzing their issue tracking systems and their version control systems.

The paper is organized as follows: Section II presents an overview of the related work in the area of the analysis and the prediction of concurrency-related defects; Section III introduces in detail our approach; Section IV discusses the results achieved; Section V presents the limitations and the threats to validity of the study; finally, Section VI draws the conclusions and presents future work.

II. Related work

In the last few years, researchers have put a lot of effort in the analysis of software projects to identify and predict some relevant properties – e.g., where defects are, how to fix them and the associated costs. A common trend in current research is investigating and trying to understand the processes by which software ages. During the years, researchers investigated the relations of various process artifacts (e.g., change history of source files, changes in the team structure, testing effort), technologies, and other human factors with software defects for bug prediction. In fact, it is well-known that process metrics are more efficient fault predictors than product metrics [27]. For instance, Nagappan *et al.* [28] in a study performed on the defect density in Windows Server 2003 used software change history (in particular, code churn measures such as changed-LOC/LOC together with dependency metrics) for predicting the bug density of each software module. Moreover, some studies have used the source code itself considered as text as an independent variable [24] [25] but LOC nearly always performs better than any other metric [43].

Another example is the study performed by Graves *et al.* [14] on a system containing 1.5 million lines of code. This work highlights that module size and other standard software complexity metrics are generally poor predictors of fault likelihood. Process metrics extracted from software change history have been used to build a weighted time damp model that considerably improved the bug prediction accuracy, if compared to previous approaches. Similar results are presented in [20], where a bug cache algorithms is used to predict future bugs at the function, method, and file level mining the related version control system and bug repository.

An interesting technique for predicting latent software bugs is called change classification. It was initially introduced in [21], where a machine learning classifier based on Support Vector Machines (SVMs) is used to determine whether a new software change is more similar to prior buggy changes or clean changes. Their classifier is trained using features (e.g., terms in the added delta source code and terms in the change log) extracted from a version archive, showing an accuracy of 78% in identifying if a file is buggy or not.

Two other interesting works focus on the impact of the software process on the defectiveness of software [40] and on the estimation of efficacy of information retrieval models for the purpose of locating bugs [34]. The latter paper also provides a comparison of five models and predicts the probability of a file to contain bugs based on its similarity with known buggy files.

A closely related research activity concerns the contextual factors influencing the transferability of bug prediction models. Nagappan *et al.* [29] investigated how different subsets of complexity metrics relate to bugs in different projects, concluding that models have good predictive performance only when trained on the same or homogeneous systems.

Good performance between releases of the same system are reported in [41] and [10], while Shatnawi *et al.* [35] report that model performances degrade when applied to later releases of a system. Although findings from individual studies on bug prediction model transferability are varied, most studies report that models perform poorly when transferred [2] [15], even if there are some exceptions [39] reporting about the transferability of the models developed on NASA data to some specific embedded domains.

Another important finding in this context is the effectiveness of the linked bugs technique in giving useful information for developing accurate defect predictive models. In [26], Moin *et al.* used bug reports information in a bug repository and the corresponding log files of the version control system (i.e., the so-called linked bugs) to train a SVM classifier. Textual information in the summary and description of bugs are used to enrich machine learning features. Experimental results prove that, given a bug report, the resulting model is able predict with a good accuracy which part of the software project is more likely to be related to the issue.

Another area we consider is the automated triage of the bugs based on the textual description available at reporting time. In [9] the authors have developed a machine learning approach based on a supervised Bayesian learner that is able to predict correctly 30% of the bugs assignments. While in [32], the authors focus on clustering the reports to identify which ones belong to the same bug.

All the previous described works focus on the analysis of sequential software projects. Unfortunately, only a few studies about bug identification and prediction in the concurrent domain have been performed. Given the complex nature of the problem and the difficulties arising from the complexity of concurrent thread interleaving analysis, most of the works focused only on studying and classifying concurrent bugs characteristics.

A comprehensive study of real world concurrency bugs is presented in [23]. By examining the bug reports and patches, corresponding source code, and programmers' discussion of four open source projects (i.e., MySQL, Apache, Mozilla, and OpenOffice), this work provides a classification of the concurrency bug patterns, occurrence conditions, fix strategies, and diagnosis processes. Another interesting work introduces a concurrent bug taxonomy aimed at identify the most common concurrent bug patterns [12].

In [13], instead of focusing on the causes of concurrency bugs, Fonseca *et al.* focus on analyzing their effects. The objective of this research is providing a new point of view that can help detecting, handling, or tolerating such defects at runtime. The two main results of the study performed on an open source project (MySQL) are the identification of latent

concurrency bugs and some useful indications for the design of reliable concurrent software systems.

A study of the applicability of sequential approaches for bug prediction model development is presented in [44]. The objective of this work is the identification of four classes of concurrency defects (i.e., Atomicity, Order, Data, and Deadlock) and the prediction of the bug quantity, type, and location from patches, bug reports, and bug-fix metrics. Two predictive models are presented and evaluated over three popular projects (i.e., Mozilla, KDE, and Apache) with encouraging results.

To the best of our knowledge, no one has investigated the possibility of an automated detection of concurrency bugs to support the bug triage phase.

III. Our investigation

We decided to focus our study on freely available open source projects with open bug tracking software and revision management system. The two projects we selected are Apache HTTP Server version 2 (HTTPD) and MariaDB (a GPL fork of MySQL). HTTPD has been chosen since it is used in many works in the bug mining research field. For the same reason, we initially wanted to include MySQL as well but we found out that the MariaDB fork is better organized and accessible as far as bug and revision management is concerned. We plan to include further projects in our study as a future work.

The aim of our investigation was to understand if machine learning techniques can be used to effectively distinguish between concurrent-related and non concurrency-related bugs. We were also interested in understanding the relevance of various bug-related information when applying these techniques.

Linked bugs [36] are those solved issues contained in a bug tracking system for which it is possible to also have access to the code modifications that led to their solution. The modifications are usually managed by a revision system. A linked bug is then a defect for which one or more links (hence the name) exist between an issue originally signaling the failure of the software system due to the bug and one or more revisions in which fixes for the bug are committed to the code base.

The following table contains an example of elements composing an actual bug report for the MariaDB system.

ID	MDEV-6833
Title	SIGSEGV on shutdown with non-default wsrep_slave_threads
Priority	MAJOR
Affects Version/s	10.1.1
Component/s	Galera
Status	CLOSED
Resolution	Fixed
Reporter	Nirbhay Choubey
Assignee	Nirbhay Choubey
Created	2014-10-03 07:04
Updated	2014-10-04 21:38
Resolved	2014-10-04 21:38
Description	Start a node with wsrep_provider; SET global.wsrep_slave_threads=3; Shutdown ...
Comment	From Nirbhay Choubey at 2014-10-03 22:36 http://lists.askmonty.org/pipermail/commits 2014-October/006695.html
Comment	From Sergei Golubchik at 2014-10-04 17:46 ok to push
Comment	From Nirbhay Choubey at 2014-10-04 21:38 https://github.com/MariaDB/server/commit/ 61d8b4a29bd6295b9db153a6ebb451346cd5bc64

ID	61d8b4a29bd6295b9db153a6ebb451346cd5bc64
Committer	Nirbhay Choubey
Date	Oct 4, 2014
Description	MDEV-6833: SIGSEGV on shutdown with non-default wsrep_slave_threads thd->variables' table_plugin & tmp_table_plugin should be set to NULL for wsrep system threads. Also made a minor change to skip checking of wsrep options if wsrep_on is not set.

The last comment a link to the web interface of the version management system is used to reference a commit fixing the bug.

Information associated to the revisions are much less structured, like the one reported in the following table.

In this case the first line of the description references the issue for which this commit is a fix. As the reader probably already noticed the issue and the revision of these examples are linking each other.

Usually, as in this example, the links between the issues in the bug tracking system and the code revisions in the version control system are not explicit (i.e. there are no fields with explicit pointers or references from one to the other). Conversely, the links must be mined from change logs and bug reports using some heuristics. To accomplish this linking task we used a traditional approach that consists in searching for specific keywords and bug IDs in change logs and bug reports. A software tool has been developed for this purpose. We compared the experimental results with other approaches such as that used in ReLink [42], a tool based on recovery algorithms which automatically learns criteria of features from a set of explicit links to recover missing links. We observed that, on our two datasets, ReLink provides results with an higher recall but a lower precision (i.e. ReLinks infers a higher number of links, but many of them are not correct).

For a linked bug, a number of information elements can be extracted with repository mining techniques:

- From the bug tracking system:
  - Bug name and description;
  - Bug metadata such as the status (solved, not a bug, etc.), the user that created the issue, the date of the initial report, etc.;
  - Discussion between users, testers and developers trying to isolate the defect.
- From the code versioning system:
  - Commit comment;
  - Revision metadata such as developer, date, etc.;
  - Modified source code.

In our machine learning perspective, mined linked bugs (our instances) are tuples with the following attributes:

- Bug id

- Bug name
- Bug description
- Bug metadata
- Bug discussion
- Bug report date
- Revision id
- Revision commit comment
- Revision metadata
- Revision creation date
- Code diffs

In our experiments linked bugs with no one-to-one bug-to-revision match (for example a bug that is incrementally solved in three revisions) are split in several one-to-one instances. In the aforementioned example, we would create three instances with the same bug linked to the three different revisions. We also experimented other approaches (as merging all the information in a single instance) and we obtained very similar results.

In order to implement a supervised learning approach, we needed to create a training set in which the ground truth had to be determined by experts analysis.

Early sampling-based investigations showed that the percentage of concurrency-related bugs for both projects is extremely low (less than 7%). This leads to two problems:

1) a very large number of linked bugs has to be examined in order to create a training set with a reasonable number of instances of the concurrency-related class;
2) the resulting training set presented a very large imbalancement (leading to well-known problems [16])

To avoid these problems, we decided not to use all the extracted linked bugs as our dataset, but we restricted to those linked bugs filtered by a plain keyword-based approach based on concurrency-related terms (such as thread, synchronization, concurrency, mutex, atomic, etc.) applied to the bug title, description, and discussion. This approach is similar to the one adopted in previous research in this area [13] [23] [44].

We randomly sampled a large number of issues not containing the aforementioned keywords and found no concurrency-related bug. We are then confident that our keyword-based method is a good starting point to identify all the bugs of this kind. On the other hand, the precision of the method is less than 1% (as suggested by the manual analysis described later). This left us with 3,336 linked bugs for Apache HTTP Server and 2,589 for MariaDB.

We served as experts to manually categorize the resulting linked bugs. In this categorization we followed the same guidelines used in [44]. Each bug has been analyzed by two experts, a third opinion has been used as tie-breaker when needed.

This resulted in 153 and 167 concurrency-related bugs for HTTPD and MariaDB respectively, as summarized in Table I.

The dataset is obviously still imbalanced but to an extent that does not prevent its use (directly, or after some specific processing) with most learners.

However, as discussed above, the keyword-based method has a very high recall. As such it does not introduce any concurrent class bias, the instances of the non-concurrent class are biased since they only include those elements that do present the concurrency-related keywords somewhere in the issue report (title, description, discussion). It may be argued that we are only making the task more difficult for a machine learning algorithm that is now called to discriminate between a concurrency bug and a non-concurrency bug that has some potentially concurrency-related term in it. Further investigation on the relevance of this bias will be performed as future work.

TABLE I
NUMBER OF BUGS CONSIDERED IN OUR STUDY.

	Concurrency related	**Non-concurrency related**	**Total**
Apache HTTP Server	153	3,183	3,336
MariaDB	167	2,422	2,589

The keyword-relevant linked bugs with the concurrent class feature added manually is our starting point to investigate the performances of different machine learning approaches. We decided not to explicitly split the dataset in training and test sets and systematically rely on cross-fold validation instead. In particular, the data reported in this paper are related to the 10-fold validation but we have also performed a 3-fold one with very similar results that are not reported in the paper.

The instances we created so far have attributes that are mostly textual (such as titles, descriptions, comments). When using machine learning for textual data it is usual to perform some pre-processing that can improve the performances of the categorization algorithms. This includes case transformation, stemming, and stop-words removal. After some experiments, we decided to perform case transformation, stemming, and stop-words removal for texts associated to issues title, description, discussion, and revisions description. No processing has been applied to the source code. As a result our dataset is now mainly composed by (processed) string data. While some machine learning algorithm (or learner as we will often refer to them in the rest of the paper) can directly cope with this type of data, most do not. We then decided to move to a representation that is more easily processable by most known learners: the bag of words. With this approach, all text is translated into a tuple of numerical values with each position in the tuple referring to a different word in the corpus (in our case, it is composed by all the text appearing in all linked bugs of the dataset).

The entries in each tuple represent the presence (variously weighted) of the corresponding word in the analyzed text. Usual weighting method are frequency, tf (term frequency, logarithmic in our case), and tfidf (term frequency-inverse document frequency) [37]. We experimented with these variants and we found out that, for our datasets, very similar performances are usually obtained with tf and tfidf, while simple frequency usually led to worst results.

Direct application of this method can easily result in a very large number of attributes (in our case in the order of tens of thousands) most of which related to words appearing only

once or twice in the corpus. Pruning is a common option in these cases; after some experiments we limited the number of words processed to the 5,000 more frequent ones for general text (bug reports, discussions, and commit messages) and to the 100 more frequent ones for the code. Please notice that this does not mean that this is the exact number of features for each data source since all the features with the same frequency as the one at the cut-off are included too. For example, setting the pruning value to 100 in the MariaDB dataset for the code results in 102 features being used since other words have the same frequency as the 100th one.

After this processing, the instances are now tuples containing all numerical values except for one nominal value, the one assigning the related linked bug to one of the two classes: concurrent and non-concurrent.

Two main aspects characterize the dataset: it is imbalanced and it contains a large number of attributes. Different learners show different degrees of susceptibility to these characters. For those that are affected, a few options exist. First of all we tested a set of learners with this basic dataset with the idea of applying some processing later and verify how that affects the various algorithms.

The following machine learning algorithms have been tested:

- **NB:** Naïve Bayes [18]
- **KN:** K-nearest neighbours classifier (K chosen using cross validation) [1]
- **C45:** C4.5 decision tree (unpruned) [33]
- **RF:** Random Forest [3]
- **SVM:** Support Vector Machine trained with sequential minimal optimization [31] [19]
- **DFE:** Bayesian network with Discriminative Frequency Estimate learning method [38]

The rationale behind this choice is to have representatives for the main classification methods that have shown effectiveness in past software repositories mining research. DFE has been introduced because it is known to provide a good Bayesian approach for feature-rich datasets like the one we are dealing with.

All learners have been tuned using common best practices. SVM has been tested with various kernels (in order to account for complex non-linear separating hyperplanes). However, the best results were obtained with a relatively simple polynomial kernel. The parameters for the resulting model have been tuned using the grid method [17]. The results we obtained with these learners are summarized in Table II.

Notice that we test the performances of the learners only with respect to the concurrent class. We do that for two main reasons:

1) We are interested in understanding if we can use machine learning techniques to identify concurrent bugs.
2) Given the imbalancement of the dataset, even a silly balancer associating any input to the non-concurrent class will have very high weighted average scores.

We are also reporting a limited amount of analysis data, specifically in this paper we focus on precision and recall (and the related F-measure). Other aspects of the learners (such as the ROC curve) have been analyzed in our tests but they were always aligned with the results expressed by the three measure we are providing here.

TABLE II
PERFORMANCE OF THE MACHINE LEARNING ALGORITHMS INVESTIGATED.

	Apache HTTP Server		
	Precision	**Recall**	**F-measure**
NB	0.166	0.614	0.261
KN	0.978	0.856	0.913
C45	0.843	0.771	0.805
RF	1	0.778	0.875
SVM	0.985	0.876	0.927
DFE	0.97	0.843	0.902

	MariaDB		
	Precision	**Recall**	**F-measure**
NB	0.231	0.671	0.344
KN	0.893	0.449	0.598
C45	0.662	0.599	0.629
RF	1	0.299	0.461
SVM	0.814	0.629	0.709
DFE	0.774	0.617	0.687

The results show that the best classifiers are those known to perform better on feature-rich datesets; given the mostly text-based nature of our dateset this was expected.

Now we want to understand if introducing mitigating methods for the dataset imbalancement and the high number of features can improve the performances of the learners.

For instance, it is well known that simple bayesian methods assume independence between all the attributes, which is almost never the case for bag of words, so we expect that eliminating correlated attributes should be beneficial for these learners. It is also known that tree-based learners, such as Random Forest, can benefit from re-balancing approaches [22].

For imbalanced datasets, there are mainly two approaches:

1) Rebalance them (by decimating the majority class or by synthetically creating new instances of the minority class).
2) Instruct the learner to give different weights to the instances of the two classes (a lower weight for those of the majority class and a higher one for the minority ones).

In the case of the large number of attributes, several feature engineering methods can be applied. The most widely adopted is attribute selection. In this case, the reduction of the number of attributes can help learners that do not perform well with a large number attributes. This helps also in reducing the computation time needed to create the predictive model. However, this last advantage can be limited when using selection algorithms that are computationally expensive. There are two main classes of attribute selection algorithms: those who analyze the performance of the learner in the selection process and those who do not use the learner.

The first class is usually very expensive from a computational point of view, since the learner runs continuously to check how it performs when changing the attributes in the dataset. Usually, that leads to computation times that are two or more orders of magnitude larger compared to the learner itself. For this reason, we did only some limited experiments with learner-aware attribute selection. In our test cases the results obtained were marginally better than those obtained with processes not using the learner. Consequently, we only used this approach in our in depth-analysis.

We performed exhaustive testing combining attribute selection (AS) with re-balancing approaches either using cost-aware version of the classifiers (CA) and/or over-sampling using the SMOTE algorithm (SM) [4]; we mostly obtained non relevant results. Table III summarized the few combinations that led to significant improvements over the basic classifiers.

TABLE III
PERFORMANCE OF THE TUNED MACHINE LEARNING ALGORITHMS INVESTIGATED.

	Apache HTTP Server		
	Precision	**Recall**	**F-measure**
NB + AS	0.259	0.771	0.388
RF + AS + CA	0.951	0.882	0.915

	MariaDB		
	Precision	**Recall**	**F-measure**
NB + AS	0.528	0.784	0.631
RF + AS + CA	0.577	0.743	0.649

The next experiment we performed is a study on the relevance of the various kinds of linked-bug information (issue description and discussion, revision comments and modified code) for classification purposes. We used the best overall learner emerged from the aforementioned tests (the SVM-based one) and we applied it to datasets obtained by selectively removing one or more components from each instance. Assuming a complete linked bug-related issue to be composed of issue-related data (I), revision-related data (R), and modified code (C), we tested the learner with all the possible combinations of I, R, and C for both datasets obtaining the results summarized in Table IV.

TABLE IV
PERFORMANCE OF THE TUNED MACHINE LEARNING ALGORITHMS INVESTIGATED.

	Apache HTTP Server		
	Precision	**Recall**	**F-measure**
I	0.958	0.902	0.929
R	0.898	0.693	0.782
C	0	0	0
C5000	0.913	0.549	0.686
I + R	0.985	0.882	0.931
I + C	0.972	0.902	0.936
R + C	0.921	0.686	0.787
I + R + C	0.985	0.876	0.927

	MariaDB		
	Precision	**Recall**	**F-measure**
I	0.833	0.599	0.697
R	0.835	0.455	0.589
C	0	0	0
C5000	0.481	0.371	0.419
I + R	0.877	0.557	0.681
I + C	0.863	0.605	0.711
R + C	0.839	0.467	0.6
I + R + C	0.89	0.581	0.703

From these results, it is obvious that the main source of discrimination is provided by issue-related information (I), followed by revision-related information (R). Modified code-related information (C), when limited to a very narrow dictionary of 100 terms, seem to carry very little discerning elements, to the point of being of no relevance on its own (F number 0) and of very limited impact when coupled with other information. When code pruning is set to 5000 (C5000) its discrimination improves but still remains under 0.7; moreover additional test we conducted (not reported in the table) show that, when combined with other information, C performs better than C5000.

Given these result, we decided to test the performance of the best learners with respect to a dataset that only includes issues' titles and descriptions. Table V shows the results we obtained using SVM and DFE.

TABLE V
PERFORMANCE OF SVM AND DFE.

	Apache HTTP Server		
	Precision	**Recall**	**F-measure**
SVM	0.978	0.889	0.932
DFE	0.967	0.758	0.85

	MariaDB		
	Precision	**Recall**	**F-measure**
SVM	0.779	0.569	0.657
DFE	0.913	0.549	0.686

The rationale behind this test is to asses the ability to construct a reasonable classifier with only the information available when a bug report is submitted. The idea here is to incrementally train a classifier with tagged bug reports (for example we can imagine to use a boolean "concurrency-related" tag applied when closing bugs) and use it for triaging newly submitted bugs. The results are promising, yet further analysis are need to understand how many tagged bug reports are need before a reliable classifier is developed.

The last test we performed tries to assess the cross-project applicability of the models produced by the learners. Specifically, we created models using SVM and DFE with one data set and applied them to the other data set. Table VI shows the obtained results.

In this experiment the same pruned dictionaries used to train the classifier with one dataset have been used to test the classifier with the other dataset; this possibly negatively

TABLE VI
CROSS-PROJECT APPLICABILITY OF THE MODELS.

Training set	Testing set	Learner	Precision	Recall	F
HTTPD	MariaDB	SVM	0.290	0.275	0.282
MariaDB	HTTPD	SVM	0.360	0.327	0.342
HTTPD	MariaDB	DFE	0.593	0.105	0.178
MariaDB	HTTPD	DFE	0.727	0.052	0.098

impacts the performances on the test dataset but it is an inherent limitation of bag of words-based approaches.

IV. DISCUSSION OF THE RESULTS

Our results show that is indeed possible to use machine learning techniques to effectively identify concurrency-related bugs.

The best overall learners in our tests have been SVM (with a polynomial kernel) and DFE, both of which do not benefit from re-balancing and feature engineering techniques. This is a relevant result: both SVM and DFE can construct categorization models with limited computational effort and the fact that they do not need further dataset filtering also eliminates the need for costly processing. In practical terms this means that, using an Intel i5 processor with two cores running at the base frequency of 1.8 GHz, a SVM-based prediction model can be created in less than 7 seconds for the MariaDB dataset and less than 13 seconds for the HTTPD dataset. A DFE-based model can be created in less than 0.1 second for both datasets.

The obtained models can easily classify more than 1,000 instances per second on the same hardware, allowing easy online processing.

Our tests also show that classification on the basis of a simple bug report can be performed with decent performances; this result combined with the possibility of easy online processing makes on-the-fly concurrency-related bug identification a concrete possibility.

One last result is related to cross-project applicability of the prediction models. Our preliminary results seem to indicate that models created with our best performing learners are useful only for the projects they are trained with. This result was totally expected since there are very few common characteristics between concurrent-related parts in the code base of the projects we tested and also limited commonalities on the format and the management of bug reports and revisions.

V. THREATS TO VALIDITY

The design, the data collection, and the analysis of the presented research has been conducted under a number of assumptions that can limit the validity of the study. In particular, the main limitations are the following:

- The study includes only two projects (Apache HTTP Server and MariaDB).
- We deal with a small number of issues, therefore the statistical significance of some of the analysis can be limited.
- The identification of the concurrency-related defects has been performed manually, therefore there could be some interpretation errors. However, to mitigate the risk, the manual check was performed by at least two authors independently.
- The software analyzers we have developed to perform the data collection and the analysis may include some bugs that prevent the identification of some relevant defects. In particular, the code analyzer considers the code as text without taking into consideration the language structure.
- There is a lack of cross-validity of the developed models since they are able to provide good results only if properly tuned on the specific project.
- There could be some biases due to: the selection of the projects to analyze (Apache HTTP Server and MariaDB), the programming language used (both C/C++), the lack of complete data about the defects and the related fixes, the selection of the non-concurrent defects only from the ones that do not include the identified keywords, the use of issues-only datasets that are extracted considering only linked bugs.

VI. CONCLUSION AND FUTURE WORK

The paper has presented an analysis of the concurrency-related defects in two popular open source projects developing also a prediction model that is able to help developers in the triage phase of the reported issues. As described in Section II, to the best of our knowledge, this is the first research work addressing this kind of problem in a quantitative way.

This result combined with the possibility of easy online processing makes on-the-fly concurrency-related bug identification a concrete possibility. This will be able to help developers of large and popular projects in the triage phase when they have to deal with a continuous flow of a large number of reported issues (G2).

Moreover, the approach can be used to perform retrospectives using all the data available after the fix of the defect slightly improving the overall performance of our models compared to the ones that we have developed based only on the information available at reporting time, as described in this paper (G1).

We have tested the performances of several algorithms and we have obtained that one of the best ones is the SVM that allowed us to achieve the following results: for Apache HTTP Server and MariaDB we have a precision of 0.985 and 0.814 and a recall of 0.876 and 0.629 when considering linked bugs (bug reports information in bug repository and the corresponding fix in the version control system) and a precision of 0.978 and 0.779 and a recall of 0.889 and 0.569 when considering only the information from bug reports. Basically, the developed models have similar performances.

However, the poor cross-validity of the developed models needs to be investigated further to analyze the possibility of developing a more general approach that is able to work with a reduced amount of training on a specific project. Moreover, the data collection can be improved extending the code analyzer

to consider the structure of the code with more advanced parsing techniques that allow a more comprehensive collection of measures.

ACKNOWLEDGMENT

The research presented in this paper has been partially funded by the ARTEMIS project EMC2 (621429) and the ECSEL project MANTIS (662189).

REFERENCES

[1] D. W. Aha, D. Kibler, M. K. Albert, "Instance-based learning algorithms", Machine Learning, 6(1), 1991.

[2] R. Bell, T. Ostrand, E. Weyuker, "Looking for Bugs in All the Right Places", International Symposium on Software Testing and Analysis, 2006.

[3] L. Breiman, "Random forests", Machine Learning, 45(1), 2001.

[4] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique", Journal of Artificial Intelligence Research, 16, 2002.

[5] P. Ciancarini, F. Poggi, D. Rossi, A. Sillitti, "Improving bug predictions in multi-core cyber-physical systems", 4th International Conference on Software Engineering for Defense Applications (SEDA 2015), Rome, Italy, 26 - 27 May 2015.

[6] P. Ciancarini, F. Poggi, D. Rossi, A. Sillitti, "Mining Concurrency Bugs", Embedded Multi-Core Systems for Mixed Criticality Summit 2016 at CPS Week 2016, Vienna, Austria, 11 April 2016.

[7] L. Corral, A. B. Georgiev, A. Sillitti, G. Succi, "A Method for Characterizing Energy Consumption in Android Smartphones", 2nd International Workshop on Green and Sustainable Software (GREENS 2013) at ICSE 2013, San Francisco, CA, USA, 20 May 2013.

[8] L. Corral, A. B. Georgiev, A. Sillitti, G. Succi, "Method Reallocation to Reduce Energy Consumption: An Implementation in Android OS", 29th ACM Symposium on Applied Computing (SAC 2014), Gyeongju, Korea, 24 - 28 March 2014.

[9] D. Cubranic, G. C. Murphy, "Automatic bug triage using text categorization", 6th International Conference on Software Engineering and Knowledge Engineering, 2004.

[10] G. Denaro, M. Pezzé, "An Empirical Evaluation of Fault Proneness Models", 24th International Conference on Software Engineering (ICSE 2002), May 2002.

[11] E. Di Bella, A. Sillitti, G. Succi, "A multivariate classification of open source developers", Information Sciences, Elsevier, Vol. 221, pp. 72 - 83, February 2013.

[12] E. Farchi, Y. Nir, S. Ur, "Concurrent bug patterns and how to test them", International Parallel and Distributed Processing Symposium (IPDPS), 2003.

[13] P. Fonseca, C. Li, R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications", 6th Conference on Computer Systems, April 2011.

[14] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, "Predicting fault incidence using software change history", IEEE Transactions on Software Engineering, 26(7), 2000.

[15] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, "A systematic literature review on fault prediction performance in software engineering", IEEE Transactions on Software Engineering, 38(6), 2012.

[16] H. He, E. A. Garcia, "Learning from imbalanced data", IEEE Transactions on Knowledge and Data Engineering, 21(9), 2009.

[17] C. W. Hsu, C. C. Chang, C. J. Lin, "A practical guide to support vector classification", Technical Report, Department of Computer Science, National Taiwan University, 2003.

[18] G. H. John, P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers", 11th Conference on Uncertainty in Artificial Intelligence, 1995.

[19] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, K. R. K. Murthy, "Improvements to Platt's SMO algorithm for SVM classifier design", Neural Computation, 13(3), 2001.

[20] S. Kim, T. Zimmermann, E. J. Whitehead Jr, A. Zeller, "Predicting faults from cached history", 29th International Conference on Software Engineering (ICSE 2007), May 2007.

[21] S. Kim, E. J. Whitehead Jr, Y. Zhang, "Classifying software changes: Clean or buggy?", IEEE Transactions on Software Engineering, 34(2), 2008.

[22] T. M. Khoshgoftaar, M. Golawala, J. Van Hulse, "An empirical study of learning from imbalanced data using random forest", 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), 2007.

[23] S. Lu, S. Park, E. Seo, Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics", ACM SIGPLAN Notices, 43(3), 2008.

[24] O. Mizuno, S. Ikami, S. Nakaichi, T. Kikuno, "Spam Filter Based Approach for Finding Fault-Prone Software Modules", 4th International Workshop on Mining Software Repositories, May 2007.

[25] O. Mizuno, T. Kikuno, "Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter", 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering, 2007.

[26] A. H. Moin, M. Khansari, "Bug localization using revision log analysis and open bug repository text categorization", International Conference on Open Source Systems (OSS 2010), May 2010.

[27] R. Moser, W. Pedrycz, G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction", 30th International Conference on Software Engineering (ICSE 2018), May 2008.

[28] N. Nagappan, T. Ball, "Use of relative code churn measures to predict system defect density", 27th International Conference on Software Engineering (ICSE 2005), May 2005.

[29] N. Nagappan, T. Ball, A. Zeller, "Mining Metrics to Predict Component Failures", 28th International Conference on Software Engineering (ICSE 2006), May 2006.

[30] E. Petrinja, Sillitti A., Succi G., Comparing OpenBRR, QSOS, and OMM Assessment Models, 6th International Conference on Open Source Systems (OSS 2010), Notre Dame, IN, USA, 30 May - 2 June 2010.

[31] J. C. Platt, "12 fast training of support vector machines using sequential minimal optimization", Advances in Kernel Methods, 1999.

[32] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports", International Conference on Software Engineering, 2003.

[33] J. Quinlan, "C 4.5: Programs for Machine Learning", Elsevier, 2014.

[34] S. Rao, A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models", 8th Working Conference on Mining Software Repositories (MSR 2011), May 2011.

[35] R. Shatnawi, W. Li, "The Effectiveness of Software Metrics in Identifying Error-Prone Classes in Post-Release Software Evolution Process", Journal of Systems and Software, 81(11), 2008.

[36] J. Śliwerski, T. Zimmermann, A. Zeller, "When do changes induce fixes?", ACM SIGSOFT Software Engineering Notes, 30(4), 2005.

[37] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval", Journal of Documentation, 28(1), 1972.

[38] J. Su, H. Zhang, C. X. Ling, S. Matwin, "Discriminative parameter learning for Bayesian networks", 25th International Conference on Machine Learning, 2008.

[39] B. Turhan, T. Menzies, A. Bener, J. Di Stefano, "On the Relative Value of Cross-Company and within-Company Data for Defect Prediction", Empirical Software Engineering, vol. 14, no. 5, pp. 540-578, 2009.

[40] E. Weyuker, T. Ostrand, R. Bell, "Using developer information as a factor for fault prediction", International Workshop on Predictor Models in Software Engineering, May 2007.

[41] E. Weyuker, T. Ostrand, R. Bell, "Do Roo Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models", Empirical Software Engineering, 13(5), 2008.

[42] R. Wu, H. Zhang, S. Kim, S. C. Cheung, "Relink: recovering links between bugs and changes". 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011.

[43] Y. Zhou, B. Xu, H. Leung, "On the Ability of Complexity Metrics to Predict Fault-Prone Classes in Object-Oriented Systems", Journal of Systems and Software, vol. 83, no. 4, pp. 660-674, 2010.

[44] B. Zhou, I. Neamtiu, R. Gupta, "Predicting concurrency bugs: how many, what kind and where are they?", 19th International Conference on Evaluation and Assessment in Software Engineering, 2015.