Reproducibility of Software Bugs

Basic Concepts and Automatic Classification

Flavio Frattini, Roberto Pietrantuono and Stefano Russo

Abstract Understanding software bugs and their effects is important in several engineering activities, including testing, debugging, and design of fault containment or tolerance methods. Dealing with hard-to-reproduce failures requires a deep comprehension of the mechanisms leading from bug activation to software failure. This chapter surveys taxonomies and recent studies about bugs from the perspective of their reproducibility, providing insights into the process of bug manifestation and the factors influencing it. These insights are based on the analysis of thousands of bug reports of a widely used open-source software, namely MySQL Server. Bug reports are automatically classified according to reproducibility characteristics, providing figures about the proportion of hard to reproduce bug their features, and evolution over releases.

1 Introduction

Software is commonly characterized by the presence of *defects*—imperfections that cause systems to improperly deliver the service they are intended for, resulting in what is called a *failure*. *Bugs* are usually meant as defects in the code, thus with a more narrow meaning than defects. Bugs have been classified according to various characteristics from the perspective of software engineering, usually with the aim of supporting product and process improvement activities by defect analysis [1]. Several schemes are available in the literature: some relevant examples are the HP

F. Frattini (✉) · R. Pietrantuono · S. Russo
Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione,
Università degli Studi di Napoli Federico II, Via Claudio, 21, 80125 Napoli, Italy
e-mail: flavio.frattini@unina.it

R. Pietrantuono
e-mail: roberto.pietrantuono@unina.it

S. Russo
e-mail: stefano.russo@unina.it

© Springer International Publishing Switzerland 2016
L. Fiondella and A. Puliafito (eds.), *Principles of Performance and Reliability Modeling and Evaluation*, Springer Series in Reliability Engineering,
DOI 10.1007/978-3-319-30599-8_21

classification [2], the IEEE 1044 classification [3], and Orthogonal Defect Classification (ODC) [4].

As software systems grow in size and complexity and are increasingly used for mission- and business-critical applications, attention is being devoted to the bug manifestation process and its features with the aim of gaining a deeper understanding of complex erroneous behaviors. Knowledge of the factors influencing the chain by which a bug is activated, propagates into the system, and reaches its external interface,[1] serves, for instance, to reproduce the exposure of subtle bugs to then locate and remove them.

Not all bugs are easily reproducible.[2] In fact, software may behave differently under apparently identical conditions. Attempting to recreate the same conditions and repeating the steps that led to an observed failure is the usual way to try to reproduce a bug. However, some bugs require rare combinations and/or relative timings of inputs, or a specific state to be reproduced; or there may be a long delay between the fault activation and the failure occurrence. In other cases, the activation of a fault makes the system traverse several error states. In all such cases it is difficult to identify the inputs and the conditions to reproduce the failure. Moreover, concurrent programs are known to suffer from the *probe effect*, an "alteration in the frequency of run-time errors observed when delays are introduced": this effect can mask synchronization errors [6]. Even when the input values and the system internal state for an observed failure are known, there are cases where the failure occurs only under specific environmental conditions, and "as the tester tries to reproduce them, the environment changes, making the failure disappear" [7].

In this chapter, we survey taxonomies and experimental studies which led to the current understanding of bugs reproducibility (Sect. 2). Then, we describe a procedure to analyze a bug repository from the reproducibility perspective (Sect. 3). The procedure is applied to thousands of bugs reported in the MySQL database management system. Results of the classification and prediction process are presented, providing insights into the process of bug manifestation and the factors influencing it (Sect. 4). This chapter ends with a brief discussion of the results (Sect. 5).

2 Studies on Bug Reproducibility

Reproducibility is the basic criterion of several bugs classifications. In the 1980s, Gray distinguished **Bohrbugs** and **Heisenbugs** [7]: Bohrbugs exhibit a deterministic behavior, hence they can be detected and removed with testing and debugging—these are also known as **hard** or **solid** ones, for which the failure occurrence is always reproducible; Heisenbugs cause transient failures, which may not manifest on a software

[1]We follow the well-established notion of *fault–error–failure chain* [5]: a *software fault* (bug) is a defect in the application code; when activated, faults cause *errors*; errors may lead to *failures*.

[2]We use the expression "bug reproducibility"—widely used in the literature—to indicate the reproducibility *of the failure* caused by the bug.

re-execution under the same input, thus appearing as “nondeterministic”. These are known as **soft** or **elusive** faults, whose activation is not systematically reproducible; they may be extremely difficult to identify through testing. Gray named the former class alluding to the physicist Niels Bohr, who developed the atomic model, and the latter class referring to the physicist Werner Heisenberg, who formulated the uncertainty principle.

Grottke and Trivedi [8, 9] recognized that the term Heisenbug had originally been coined in the 1960s by Lindsay (while working with Gray), referring to “bugs in which clearly the behavior of the system is incorrect, and when you try to look to see why it’s incorrect, the problem goes away”; this is a different definition than the one published later by Gray. The class of bugs defined based on the notion of non-determinism is not the same as soft faults (as claimed by Gray).

The nomenclature that Grottke and Trivedi revised in the past decade introduces the category of **Mandelbugs** in lieu of Heisenbugs: by alluding to the mathematician Benoit Mandelbrot and his work on fractal geometry, the name somehow suggests a form of chaotic system behavior [8]. Unlike Heisenbugs, Mandelbugs are defined in terms of *inherent* faults properties, that is, faults *able* to cause failures which are not systematically reproducible.[3]

Four factors of complexity in the failure reproduction process are pinpointed by Grottke, Nikora, and Trivedi as responsible for a bug to be classified as Mandelbug [10]: (i) a time lag between the fault activation and the failure occurrence; (ii) interactions of the software application with hardware, operating system, or other applications running concurrently on the same system; (iii) influence of the timing of inputs and operations; (iv) influence of the sequencing of operations.

Several studies show that Bohrbugs are much more common than Mandelbugs: for instance, 463 out of 637 bugs are classified as Bohrbugs in [11]; 547 over 852 bugs are considered as always reproducible in [12]. Other studies examine bugs according to characteristics attributable to Bohr- or Mandelbugs (factors such as concurrency, resource management), but adopting a different terminology [13–15].

Mandelbugs are often related to unusual hardware conditions (rare or transient device faults), limit conditions (out of storage, counter overflow, lost interrupt, etc.,), or race conditions [7]. Changing the environment and removing the chaotic state—i.e., resetting the program to a proper state—is likely to enable the application to work. This explains why and how some fault tolerance methods work. For example, checkpointing is a technique that periodically saves a snapshot of an application in permanent storage, enabling it to restart after a failure from an internal state that should allow the proper execution. Similarly, replicating the execution in two

[3]Gray states: “Heisenbug may elude a bugcatcher for years of execution. The bugcatcher may perturb the situation just enough to make it disappear. This is analogous to Heisenberg’s Uncertainty Principle in physics.” Indeed, Heisenberg ascribed the uncertainty principle to the disturbance triggered by the act of measuring (observer effect). However, this argument is recognized to be misleading by modern physicists: the principle states a fundamental property of conjugate entities; it is not a statement about the observational success of the technology. Curiously, Mandelbugs are closer than Heisenbugs to the principle these were originally meant to resemble.

different environments can result in the proper execution of a replica even if the other fails.

A third category of bugs introduced in [8] are **aging-related bugs** (ARBs): they cause the accumulation of errors in the running application or in the system-internal environment, which result in an increased failure rate and/or degraded performance. They are viewed as a class of *Mandelbugs*. It is worth noting that ARBs are hard to reproduce due to both the system internal state and the time necessary for the failure to manifest. They might be easy to activate, but the time they take to manifest themselves as failures make them hard to observe during testing.

In [16–20], we focused on the aging problem and on aging-related bugs, distinguishing memory-related problems (e.g., leaks), storage-related (e.g., fragmentation), wrong management of other resources (e.g., handles, locks), and numerical errors, observing an impact of approximately 5 %.

Other researchers focused on the ephemerality of bugs. Chandra and Chen [21] distinguish environment-independent and environment-dependent bugs, and further classify the latter as transient and non-transient. **Environment-independent bugs** occur independently of the operating environment; if a bug of this kind occurs with a specific workload, it will always occur with that workload. **Environment-dependent bugs** depend on the operating environment: the subset of **transient bugs** may appear only in some executions; on the contrary, **non-transient bugs** occur always, in a specific environment. Environmental bugs are also examined in [22], where authors focus on how to reproduce transient bugs by varying factors of the execution environment. Clearly, there is an overlap between Mandelbugs/Heisenbugs and environment-dependent transient bugs: in both cases the main characteristics are the apparent aleatory occurrence and the difficulty of reproduction. In [23], environment-dependent bugs are categorized to conjecture a possible fault injection framework for emulating environmental influence on systems failure.

Concurrency bugs are discussed in [13]: the authors select randomly 105 such bugs from the repositories of 4 open-source applications, showing that their reproducibility is very hard due to the large number of variables involved or to the dynamics of memory accesses. Fonseca et al. [24] consider concurrency bugs from MySQL Server to understand their effects and how they can be detected or tolerated. In this case, it is shown that concurrency bugs are likely to remain latent and to corrupt data structures, but only later cause actual failures. In [14], it is shown that concurrency bugs in operating system code commonly requires more effort to be identified. Moreover, the analysis reveals that semantic bugs are the dominant root cause and, as software evolves, their number increases while memory-related bugs decrease. *Aging-related concurrency bugs* are shown in [18] to cause performance decrease over time in a hard-to-predict way, and the failure rate does not depend on the instantaneous and/or mean accumulated work.

Performance bugs are analyzed by Nistor et al. [25]. It is shown that: their fixing may introduce new functional bugs; they appear more difficult to fix than nonperformance bugs; most performance bugs are discovered through code reasoning rather than because of users experiencing failures due to bugs. In [26], some rules are extracted from a set of bugs in order to identify performance problems in MySQL, Apache, and Mozilla applications. *Security bugs* are studied in [27] with reference to the Firefox software. It is shown that they require more experienced debuggers and their fixes are more complex than the fixes of other kinds of bugs. For open source software, an empirical study based on the automated analysis of 29,000 bugs [28] shows that most bugs are semantic and that bugs related to memory are still the major component, despite the recent introduction of detection tools.

These studies examine several high-level factors that can be (directly or indirectly) attributed to the bug manifestation process, considering aspects such as concurrency, memory, timing, interaction with the operating system or other applications, performance, security, resource leaks, wrong error handling. Overall, the literature highlights the relevance of the topic, but there is insufficient knowledge of essential bug reproducibility characteristics. With respect to other bug characteristics (e.g., detection or closing times, bug location, i.e., source code, fixing commits, severity, etc.), that are more amenable to be analyzed automatically, a relevant problem is the absence of approaches to automatically distinguish bugs according to some reproducibility characteristic. In the following, we attempt to address this issue by proposing an automatic classification from reports, in order to enable future analyses on wider datasets, so as to improve the knowledge about bug reproducibility similarly to other defect analysis research areas.

3 Analysis of Bug Reproducibility

We describe a procedure to analyze bug reports from the *reproducibility* perspective, i.e., considering how they can be exposed and reproduced. We refer to a system model where we distinguish the *application* under analysis, its *execution environment*, the *workload*, and the *user* submitting it. The analysis is meant to discriminate bugs depending on whether, under a given workload, they are *always reproducible*, or *not always reproducible*, i.e., they may occur or not depending on the state of the execution environment. The latter is considered as the hardware resources where the application is deployed (processors, I/O devices, network), and software running concurrently on each node—including operating systems, middleware layers, virtualization layers, and other applications sharing the hardware resources.

The user (not necessarily a human) interacts with the application by submitting workload requests and getting the results. We assume a workload request represented as a generic request for service (e.g., a query to a DBMS), characterized by a type (e.g., query type, like INSERT), and by a set of input parameters (e.g., values of an INSERT), in turn characterized by a type and a value. To accomplish a well-defined task, the user can submit a sequence of serial/concurrent requests. We denote with *environment* the union of the *execution environment* and the *user*.

According to this model, we define two categories of bug manifestation.

- **Workload-dependent (WL)**: *the bug manifestation is* "workload-dependent" *if resubmitting (at most a subset of) the workload requests that caused a failure always produces the same failure, for every valid state of the environment (i.e.,* *for every state of the environment in which the traversed application states are allowed to occur) and for every admissible user inputs' timing/ordering in each request of the sequence*. In this sense, we talk about *always reproducible bugs*.

  - An **example** of this kind of bug is the bug 13894 of MySQL Server,[4] which reports "Server crashes on update of CSV table" and also includes the sequence of statements that crashes the server. The reporter specifies that every time the sequence is repeated the MySQL Server crashes.

- **Environment-dependent (ENV)**: *the bug manifestation is* "environment-dependent" *if resubmitting (at least a subset) of workload requests that caused a failure, there exist at least one (valid) state of the environment or user inputs' timing/ordering*[5] *causing the same failure to not be reproduced*. Note that, unlike Mandelbugs, these do not include complex but "deterministic" bugs, namely those bugs requiring a very complex workload but that, under such workload, always reappear: in this categorization, such bugs fall in the WL category. We talk about *not-always-reproducible bugs*.

  - As an **example** consider the bug 18306 of MySQL Server "MySQL crashes and restarts using subquery"; the report is about a delete operation; the reporter specifies that "The DELETE query works X times and then mysql crashes." Thus, the bug does not manifest every time a specific load is submitted to the system, further conditions are to be forced in order to reproduce the bug, instead.

The analysis aims at providing insights into the presence of workload-dependent or environment-dependent bugs and of their evolution over several software versions. The following procedure is followed: (i) first, *manual classification* of bugs as either WL or ENV is performed, by inspecting bug reports of the target software; (ii) then, we build predictors for *automatic classification* that takes bug reports as input, process the text contained in the report by text mining techniques, and then automatically classify the report as either WL or ENV; (iii) based on the predicted values, further insight about WL versus ENV bugs on an enlarge dataset and on various versions of the software is obtained; (iv) the most discriminating textual features are also examined, so as to figure out the characteristics of a bug more related to the bug manifestation type. The analysis steps are summarized in the next sections.

3.1 Manual Classification

Each problem report is manually inspected to check it documents a real and unique bug: documentation and build issues, problems turning out to be operator errors,

[4]MySQL Bugs—https://bugs.mysql.com.

[5]*Valid* means admissible, compatible environment state with reference to the input requests; in the case of user, it means that the same workload request(s) could be submitted in different timing/ ordering producing the same result.

and requests for software enhancements (not erroneously marked as such) are put in a NOT_BUG class and discarded. Then, reports containing insufficient details to classify the bug (e.g., a bug closed as soon as it was reported, corrected by a new minor release) are assigned to the class UNKNOWN, and discarded.

The remaining reports are searched for the following indications:
(i) what inputs are required (the failure-causing workload); reports often provide it in the test case and/or in the steps to repeat the failure occurrence;
(ii) what is the application and the environment configuration;
(iii) if the corresponding failure is observed to be always repeatable (i.e., workload-dependent) or not (i.e., environment-dependent) by the bug reporter or assignee;
(iv) in the case not always reproducible, what are the conditions hiding the bug manifestation (i.e., if they are related to the execution environment, or to particular user actions, such as timing of inputs).

Based on this, we assign a bug report to the workload-dependent (WL) or environment-dependent (ENV) classes. It is wort noting that the manual analysis is based exclusively on fixed bugs (as in [12, 15, 16]), since, for bugs that have not yet been fixed, the reports may contain inaccurate or incomplete information. While this allows relying on more stable information, results will not refer to non-closed bugs, which could, in principle, have different patterns of WL or ENV bugs. Moreover, although results of the manual classification are cross-checked by the authors, we cannot exclude, as any paper where manual inspection is needed, possible classification mistakes that can affect the results.

3.2 *Automatic Classification*

The bug classification aims at automatically assigning a bug to a class by analyzing its report. It consists of two steps: text processing and classification.

3.2.1 Text Processing

The automatic classification is carried out by means of predictors, using bug report textual description as input features. Some preliminary steps are required to render text suitable for processing by classifiers. Specifically, in text mining, each term occurring in the document is a potential dimension; to avoid dealing with a useless large number, we apply common reduction methods [29, 30]:

1. **Tokenization**: a textual string is divided into a set of tokens; a token is a block of text considered as a useful part of the unstructured text (it often corresponds to a word, but it might also be an entire sentence). Tokenization includes filtering out meaningless symbols, like punctuation, brackets, and makes all letters lowercase.

2. **Stop-word removal**; this consists of removing the terms such as propositions, articles, conjunctions, which do not convey much useful information and may appear frequently—thus biasing the classification algorithms.
3. **Stemming**: reducing common words to a single term, e.g., “computerized”, “computerize”, and “computation” are all reduced to “compute”.

3.2.2 Classifiers

To classify the bugs, we adopt two classifiers widely used in the literature of defect prediction, namely *Naïve Bayes* and *Bayesian Network*.

A Naïve Bayes (*NB*) classifier estimates the a posteriori probability of the hypothesis H to be assessed (e.g., “bug is ENV”), that is the probability that H is true given an observed evidence E. This is given the probability to observe E under the hypothesis H multiplied by the a priori probability of the hypothesis H (i.e., when no evidence is available) over the probability of evidence E:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}. \tag{1}$$

The evidence E consists of *features* used to classify, namely the *attributes* of the instances to classify, which are extracted through the mentioned text mining technique. A fundamental assumption of a Naïve Bayes classifier is that each feature E_i is conditionally independent of any other feature E_j, $j \neq i$. Under this assumption, the a posteriori probability can be obtained as:

$$P(H|E) = \left[\prod_i P(E_i|H)\right] \frac{P(H)}{P(E)}. \tag{2}$$

This assumption is apparently oversimplifying, since features usually exhibit some degree of dependence among each other. Nevertheless, the Naïve Bayes classifier performs well even when this assumption is largely violated [31].

A Bayesian network (*BayesNet*) is a directed acyclic graphical model representing a set of random variables and their conditional dependency (graph nodes and edges, respectively). A conditional dependency exists between two nodes if the corresponding random variables are not conditionally independent. It is assumed that:

$$P(\text{node}|\text{parents plus any other nondescendants}) = P(\text{node}|\text{parents}). \tag{3}$$

The joint probability distribution for a set of random variables $X_1, \ldots, X_n$ is:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|X_{i-1}, \ldots, X_1) = \prod_{i=1}^{n} P(X_i|X_i\text{'s parents}). \tag{4}$$

Equation 4 is used to compute the probability of a hypothesis H represented by a node of the network, given the conditional probability distributions of each node, and given a set of observed values.

Classifiers are evaluated by means of the common metrics *precision*, *recall*, and *F-measure*. We adopt a *k-fold cross-validation*, with $k = 3$ and 10 repetitions: the set is split into three disjoint sets, two of which are used for training, and the third one for testing. Accuracy metrics are then calculated. The procedure is repeated until each subset has been used as test set. Accuracy metrics of each step are averaged. All the steps are repeated ten times; each time the three sets are generated by randomly selecting reports from the set. Finally, the metrics values of each repetition are averaged to obtain the final accuracy evaluation.

For each data sample in the test set, the predicted class is compared with the actual class of the sample. Given a target class (e.g., ENV), samples of the test set belonging to the *target class* are *true positives* if they are correctly classified, and are *false negatives* otherwise. Similarly, samples belonging to the other class (i.e., WL) are denoted as *true negatives* if they are correctly classified, and as *false positives* otherwise. From these, we compute:

- **Precision (Pr)**: Percentage of *true positives* (TP) that are classified as belonging to the target class (*true positives* and *false positives* (FP)):

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}). \tag{5}$$

- **Recall (Re)**: Percentage of *true positives* that actually belong to the target class (*true positives* and *false negatives* (FN)):

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN}). \tag{6}$$

- **F-measure (F)**: Harmonic mean of *precision* and *recall*:

$$F\text{-measure} = (2 \cdot \text{Pr} \cdot \text{Re})/(\text{Pr} + \text{Re}). \tag{7}$$

The higher the precision and the recall (ideally, $\text{Pr} = \text{Re} = 1$), the higher the quality of the predictor, since it avoids false positives and false negatives.

4 Case Study: Analysis of MySQL Bugs

The procedure described in Sect. 3 is here applied to the MySQL Data Base Management System. This is chosen as case study since it is a modern complex software system widely adopted in business-critical contexts, and for which detailed bug reports are publicly available.

We use bug reports related to *MySQL Server* version 5.1 for manual classification and training. Then, the prediction is applied to bug reports from other versions of the software. We consider both versions preceding the 5.1 (MySQL Server 4.1 and 5.0) and versions following the 5.1 (5.5 and 5.6). The aim is to figure out if there is a trend for the types of bugs over various versions, and to understand which modifications in the software are likely the cause of such a trend, if any.

4.1 Manual Classification

As a preliminary step, the following types of reports are excluded manually from subsequent inspection:

- reports not marked as *closed*, so as to proceed with descriptions only of solved bugs, relevant for the analysis;
- *duplicate* bugs, that is, from the description we deduced that the reported problem was caused by the same bug of another report already classified;
- bugs marked as *enhancement* or *feature request* in the "severity" field.

This step produces a set of 560 bug reports, which are manually inspected according to the steps reported in Sect. 3.1, considering the following reports' sections:

- the textual description of the steps to repeat the failure;
- the textual discussion and comments of developers/users working on that bug;
- the final patch that has been committed, along with the description note in the change log;
- the attached files (e.g., test cases, environment configuration files).

The manual inspection identifies:

- 402 workload-dependent (WL) bugs;
- 86 environment-dependent (ENV) bugs;
- 44 reports classified as NOT_BUG;
- 28 reports classified as UNKNOWN, because thy contained insufficient details.

We have that 82 % of classified bugs are WL, and the remaining 18 % are ENV. As in other studies, workload-dependent bugs are the large majority [10–12]. Nevertheless, the minority of environment-dependent bugs are those hard to reproduce and to fix.

4.2 Automatic Classification

The automatic classification is performed on the set of bugs manually categorized as WL or ENV; it includes the two steps of text processing and classification.

4.2.1 Text Processing

Text processing of reports is performed to identify the features and their occurrence. To identify the features the three operations of tokenization, stop-word removal, and stemming are performed, as described in Sect. 3.2. 21,477 features are identified. For each of them it is counted the number of occurrences in each report.

4.2.2 Classifiers

The classifiers are trained and their accuracy is evaluated by means of threefold cross-validation. Results on the Naïve Bayes and Bayesian Network classifiers are reported in Table 1.

Results show that the Bayesian Network classifier presents a lower precision than the Naïve Bayes classifier, but recall is better. Overall, the F-measure reveals that the Bayesian Network is slightly better than the Naïve Bayes classifier.

We also report the confusion matrices, which show, for each class, the number of bugs correctly classified and the incorrectly classified instances. Tables 2 and 3 are related to the Naïve Bayes the Bayesian Network classifiers, respectively.

The tables show that most of the incorrect classifications are related to environment-dependent bugs. In the case of the Naïve Bayes classifier, 73 % of environment-dependent bugs were not correctly classified. This may be due to the reduced number of examples for training. However, the classification improves when using the Bayesian Network, both for WL bugs and for ENV bugs.

It is also interesting to consider which features are the most discriminating. The classifiers predict the type of a bug (i.e., workload-dependent or environment-dependent) based on the terms used in its report. During the training, the probability, for each feature, that a bug is of a certain type given that the feature appears in the description is computed. In Table 4, we report the most significant features for the prediction process.

Table 1 Results of the training

	Class	Precision	Recall	F-measure
Naïve Bayes	WL	0.94	0.81	0.87
	ENV	0.41	0.73	0.53
	Weighted	0.86	0.80	0.82
Bayesian Network	WL	0.90	0.90	0.90
	ENV	0.44	0.46	0.45
	Weighted	0.83	0.83	0.83

Table 2 Confusion matrix of Naïve Bayes classifier

Correct	Incorrect	
113	27	WL
7	19	ENV

In some cases there is just a root indicating a set of words; as an example, *concurr* implies that there may be words such as concurrent, concurrency, etc. Note that most features are commonly linked to hard-to-reproduce conditions, such as memory

Table 3 Confusion matrix of Bayesian Network

Correct	Incorrect	
125	15	WL
14	12	ENV

Table 4 Most significant features for prediction

Feature		
Thread	Memory	Concurr
Deadlock	Alloc	Wait
Run	Race	Start
Schedul	Valgrind	

issues, deadlocks, or race conditions. Valgrind is a tool for solving memory issues; thus, the feature with the same name is also related to memory issues.

It is worth noting that only 259 features, over the total 21,477, are actually used by the classifiers.

4.3 Prediction Results

The prediction of bug types is performed for the two versions preceding the one used for the training, and for the two following it: that is for MySQL Server versions 4.1, 5.0, 5.5, and 5.6. Results are shown in Figs. 1 and 2 for the Naïve Bayes classifier and the Bayesian Network, respectively. In the figures, results achieved by means of reports' manual inspection (version 5.1) are marked differently.

As expected, workload-dependent bugs are more than environment-dependent ones. This is even more evident using the Bayesian network. For preceding versions, just 6.7 % of bugs are ENV. The small number of environment-dependent bugs is

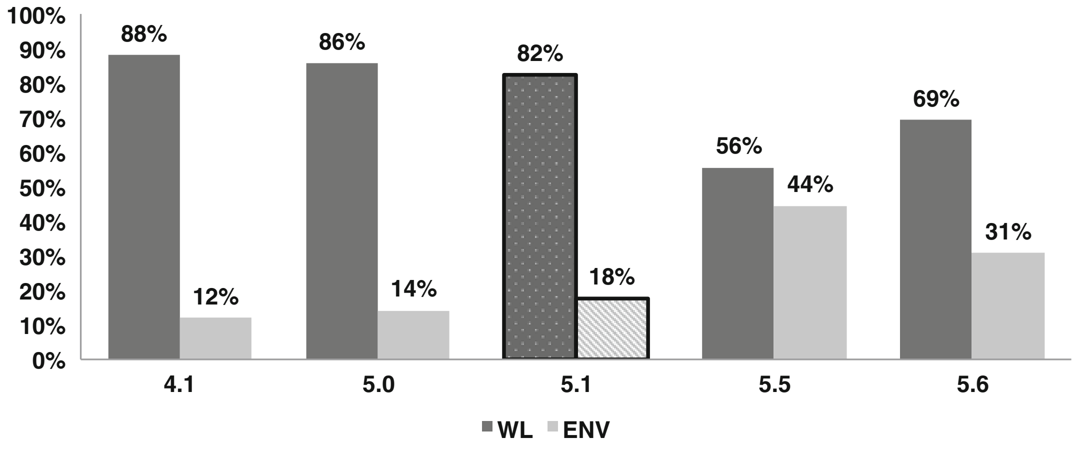

Fig. 1 Results on bug type prediction with *Naïve Bayes* classifier

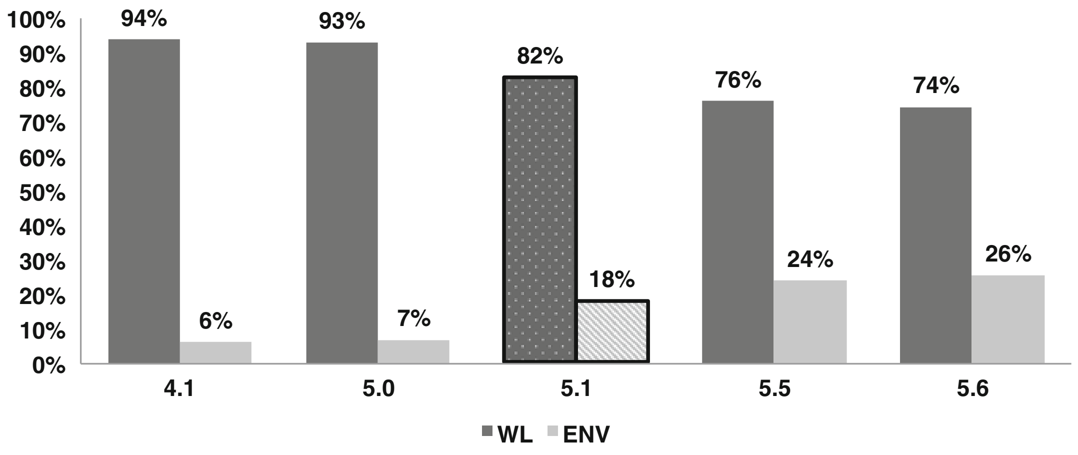

Fig. 2 Results of bug type prediction with *Bayesian Network*

probably related to the type of software considered. The DBMS, in its early versions, is not made up of many subsystems, each influencing the condition for which a fault is activated. Environment-dependent bugs are more common for operating systems, instead [12].

It is also worth noting the increasing trend of environment-dependent bugs' percentage over versions, in line with the common opinion that the prevalence of simple bugs decreases with time [12]. While in version 4.1 just 6 % of the bugs are predicted as ENV, this percentage increases up to 26 % for version 5.6. This may be due to the greater complexity starting from version 5.5. In the documentation, it is specified that one of the main changes in later versions, with respect to previous ones, consists in multiple background I/O threads used to improve I/O performance. The status of such threads may represent an environmental condition hard to reproduce. Interestingly, we also found that *thread* is the most discriminating feature.

5 Discussion and Conclusion

Reproduction of software failures to locate and fix the corresponding bugs is not an easy task, but it is very important to make software reliable. We discussed the problem of bug manifestation and introduced the two classes of workload-dependent and environment dependent bugs. For the former class, by resubmitting the same workload requests that caused a failure, the same failure can be produced. For the latter, there exists at least one (valid) state of the environment or user inputs' timing/ordering that may render the same failure unreproducible even when resubmitting the same workload request that caused the failure.

The analysis of the literature and the manual inspection of 560 bug reports showed that environment-dependent bugs are, commonly, in small number with respect to workload-dependent bugs. They are more difficult to fix, however. Also, studies in the scientific literature are usually related to few hundreds of bugs, given the difficulty of inspection by hand. Thus, we applied text mining techniques and Bayesian classifiers in order to automate the classification process.

Results from classification models show that automatic classification can be performed with an F-measure up to 83 %, but with environment-dependent bugs being more difficult to discriminate. Future works will target the improvement of environment-dependent bug discrimination by exploiting other sources of information (e.g., complexity metrics). Results from the prediction of MySQL reports confirm that, for large datasets, workload-dependent bugs are more numerous. Nevertheless, it is worth noting that the number of environment-dependent bugs presents an increasing trend from one version to another. This may be due to the addition of more software components that may cause the necessity for particular conditions of each of them in order to produce a failure.

More bug reports from different kinds of software (e.g., web servers, operating systems, apart from DBMS) should be analyzed in order to understand the trend of the two types of bugs. Also, how these and other classifiers can be improved in order to increase the quality of the prediction.

References

1. Carrozza G, Pietrantuono R, Russo S (2014) Defect analysis in mission-critical software systems: a detailed investigation. J Softw Evol Process 27(1):22, 49
2. Grady RB (1992) Practical software metrics for project management and process improvement. Prentice Hall, Englewood Cliffs
3. IEEE Computer Society IEEE Standard Classification for Software Anomalies, IEEE Std 1044–2009
4. Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong M-Y (1992) Orthogonal defect classification-a concept for in-process measurements. IEEE Trans Softw Eng 18(11):943–956
5. Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33
6. Gait J (1986) A probe effect in concurrent programs. Softw Pract Exp 16(3):225, 233
7. Gray J (1985) Why do computers stop and What can be done about it? Tandem Tech Report TR-85.7
8. Grottke M, Trivedi KS (2005) A classification of software faults. In: Supplemental proceedings 16th IEEE international symposium on software reliability engineering (ISSRE), pp 4.19–4.20
9. Grottke M, Trivedi KS (2007) Fighting bugs: remove, retry, replicate, and rejuvenate. Computer 40(2):107–109
10. Grottke M, Nikora A, Trivedi KS (2010) An empirical investigation of fault types in space mission system software. In: Proceedings IEEE/IFIP international conference on dependable systems and networks (DSN), pp 447–456
11. Chillarege R (2011) Understanding Bohr-Mandel bugs through ODC triggers and a case study with empirical estimations of their field proportion. In: Proceedings 3rd IEEE international workshop on software aging and rejuvenation (WoSAR), pp 7–13
12. Cotroneo D, Grottke M, Natella R, Pietrantuono R, Trivedi KS (2013) Fault triggers in open-source software: an experience report. In: Proceedings 24th IEEE international symposium on software reliability engineering (ISSRE), pp 178–187
13. Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput Architect News 36(1):329–339

14. Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical Softw Eng 19(6):1665–1705
15. Carrozza G, Cotroneo D, Natella R, Pietrantuono R, Russo S (2013) Analysis and prediction of mandelbugs in an industrial software system. In: Proceedings IEEE 6th international conference on software testing, verification and validation (ICST), pp 262–271
16. Cotroneo D, Natella R, Pietrantuono R (2013) Predicting aging-related bugs using software complexity metrics. Perform Eval 70(3):163–178
17. Bovenzi A, Cotroneo D, Pietrantuono R, Russo S (2011) Workload characterization for software aging analysis. In: Proceedings 22nd IEEE international symposium on software reliability engineering (ISSRE), pp 240–249
18. Bovenzi A, Cotroneo D, Pietrantuono R, Russo S (2012) On the aging effects due to concurrency bugs: a case study on MySQL. In: Proceedings 23rd IEEE international symposium on software reliability engineering (ISSRE), pp 211–220
19. Cotroneo D, Natella R, Pietrantuono R (2010) Is software aging related to software metrics? In: Proceedings IEEE 2nd international workshop on software aging and rejuvenation (WoSAR), pp 1–6
20. Cotroneo D, Orlando S, Pietrantuono R, Russo S (2013) A measurement-based ageing analysis of the JVM. Softw Test Verif Reliab 23:199–239
21. Chandra S, Chen PM (2000) Whither generic recovery from application faults? A fault study using open-source software. In: Proceedings international conference on dependable systems and networks (DSN), pp 97–106
22. Cavezza DG, Pietrantuono R, Russo S, Alonso J, Trivedi KS (2014) Reproducibility of environment-dependent software failures: an experience report. In: Proceedings 25th IEEE international symposium on software reliability engineering (ISSRE), pp 267–276
23. Pietrantuono R, Russo S, Trivedi K (2015) Emulating environment-dependent software faults. In: 2015 IEEE/ACM 1st international workshop on in complex faults and failures in large software systems (COUFLESS), pp 34–40
24. Fonseca P, Cheng L, Singhal V, Rodrigues R (2010) A study of the internal and external effects of concurrency bugs. In: Proceedings international conference on dependable systems and networks (DSN), pp 221–230
25. Nistor A, Jiang T, Tan L (2013) Discovering, reporting, and fixing performance bugs. In: Proceedings 10th conference on mining software repositories (MSR), pp 237–246
26. Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. In: Proceedings 33rd ACM SIGPLAN conference on programming languages design and implementation (PLDI), pp 77–88
27. Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on Firefox. In: Proceedings 8th conference on mining software repositories (MSR), pp 93–102
28. Li Z, Tan L, Wang X, Lu S, Zhou Y, Zhai C (2006) Have things changed now?: an empirical study of bug characteristics in modern open source software. In: Proceedings 1st workshop on architectural and system support for improving software dependability (ASID), pp 25–33
29. Lamkanfi A, Demeyer S, Soetens QD, Verdonck T (2011) Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings 15th European conference on software maintenance and reengineering (CSMR), pp 249–258
30. Menzies T, Marcus A (2008) Automated severity assessment of software defect reports. In: Proceedings IEEE international conference on software maintenance (ICSM), pp 346–355
31. Domingos P, Pazzani M (1997) On the optimality of the simple Bayesian classifier under zero-one loss. Mach Learn 29(23):103–130