Detecting Missing Information in Bug Descriptions

Oscar Chaparro[1], Jing Lu[1], Fiorella Zampetti[2], Laura Moreno[3]
Massimiliano Di Penta[2], Andrian Marcus[1], Gabriele Bavota[4], Vincent Ng[1]
[1]The University of Texas at Dallas, USA – [2]University of Sannio, Italy
[3]Colorado State University, USA – [4]Università della Svizzera italiana, Switzerland

ABSTRACT

Bug reports document unexpected software behaviors experienced by users. To be effective, they should allow bug triagers to easily understand and reproduce the potential reported bugs, by clearly describing the Observed Behavior (OB), the Steps to Reproduce (S2R), and the Expected Behavior (EB). Unfortunately, while considered extremely useful, reporters often miss such pieces of information in bug reports and, to date, there is no effective way to automatically check and enforce their presence. We manually analyzed nearly 3k bug reports to understand to what extent OB, EB, and S2R are reported in bug reports and what *discourse patterns* reporters use to describe such information. We found that (i) while most reports contain OB (*i.e.*, 93.5%), only 35.2% and 51.4% explicitly describe EB and S2R, respectively; and (ii) reporters recurrently use 154 discourse patterns to describe such content. Based on these findings, we designed and evaluated an automated approach to detect the absence (or presence) of EB and S2R in bug descriptions. With its best setting, our approach is able to detect missing EB (S2R) with 85.9% (69.2%) average precision and 93.2% (83%) average recall. Our approach intends to improve bug descriptions quality by alerting reporters about missing EB and S2R at reporting time.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Maintaining software**;

KEYWORDS

Bug Descriptions Discourse, Automated Discourse Identification

1 INTRODUCTION

Bug reports are meant to collect relevant information about the bugs that users encounter when using software. The information provided in such reports is intended to help developers diagnose and remove software bugs [78]. While much of the information in bug reports is structured, the main content of a bug report is unstructured, that is, expressed in natural language [26, 63, 78]. Unstructured natural language content produced by reporters includes the description of software's (mis)behavior (*i.e.*, Observed Behavior or OB), the steps to reproduce the (mis)behavior (*i.e.*, Steps to Reproduce or S2R), and the software's Expected Behavior (EB). Previous research indicates these three pieces of information to be highly important for developers when triaging and fixing bugs [41, 78]. While considered extremely useful, reporters do not always include OB, EB, and S2R in their bug reports. Recently, developers from more than one thousand open source projects signed and sent a petition to GitHub remarking that "... issues are often filed missing crucial information like reproduction steps ..." [1]. In addition, researchers found that textual descriptions in bug reports are often incomplete, superficial, ambiguous, or complex to follow [30, 41, 51, 71, 77, 78].

The lack of important information in bug reports is one of the main reasons for non-reproduced bugs [30], unfixed bugs [77], and additional bug triage effort [19], as developers have to spend more time and effort understanding bug descriptions or asking for clarifications and additional information [19, 30]. Low-quality bug reports are also likely to gain low attention by developers [31]. As indicated by developers, absent and wrong information in bug reports is the predominant cause for delays on bug fixing [78].

One of the main reasons for incomplete information in bug descriptions is the inadequate tool support for bug reporting [1, 30, 78]. In the aforementioned GitHub petition [1], developers called for improvements to GitHub's technology to ensure that essential information is reported by users. This problem extends to other bug tracking systems. Most of these systems capture unstructured natural language bug descriptions through web forms without any content verification or enforcement. Some bug tracking systems (*e.g.*, Bugzilla in the Mozilla Firefox project [65]) provide semi-structured reporting of natural language information, using predefined text templates that explicitly ask for OB, EB, and S2R. Such a solution is insufficient to address the problem, as it does not guarantee that reporters will provide this information as expected.

Very little research has been done on detecting the presence/absence of OB, EB, or S2R in bug descriptions. Most of the approaches proposed in the literature are meant to detect other types of information, such as source code snippets or stack traces [14, 17, 58, 60, 73]. The few that detect OB, EB, or S2R [17, 26, 78] rely on keyword matching, such as *"observed behavior"* to detect OB, or basic heuristics, such as enumerations/itemizations identification to detect S2R. Unfortunately, while simple and straightforward, these approaches are suboptimal in accurately detecting such content, as they lead to an excessive number of undetected cases (*i.e.*, false negatives) [26].

The goal of our research is two-fold: (i) to understand *to what extent* and *how* reporters describe OB, EB, S2R in bug descriptions; and (ii) to develop and validate an approach to automatically identify if bug reports miss such contents. Our conjecture is that *reporters use a limited vocabulary and a well-defined set of discourse patterns when describing OB, EB, or S2R.* If true, then we can automatically detect the presence or absence of these patterns with high

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106285

accuracy. The converse situation would mean that the automatic analysis of unstructured bug descriptions would be impractical. To the best of our knowledge, no existing research validates or invalidates our conjecture. The work on bug description analysis has mainly focused on investigating linguistic properties of bug report titles [39], identifying frequently asked questions [19], investigating unwanted behavior types [24], and studying the structure of bug reports [26, 63, 78]. Other work has focused on identifying linguistic patterns in other software engineering sources, such as development e-mails [27] or app reviews [55]. Little is known about the discourse that reporters use to describe software bugs.

We manually analyzed 2,912 bug reports from nine systems and found that, while most reports contain OB (*i.e.*, 93.5%), only 35.2% and 51.4% explicitly describe EB and S2R, respectively. In addition, to verify our conjecture, we analyzed sentences and paragraphs of a subset of 1,091 bug report descriptions by using an open coding procedure [49]. We found that reporters recurrently use 154 discourse patterns to describe OB, EB, and S2R, which means that such contents can be automatically detected. Based on our findings, we developed an automated approach for detecting missing EB and S2R in bug report descriptions (as they are more likely to be missing), called DeMIBuD–**De**tecting **M**issing **I**nformation in **Bu**g **D**escriptions. We developed three versions of DeMIBuD based on regular expressions, heuristics and Natural Language Processing (NLP), and Machine Learning (ML). We empirically evaluated DeMIBuD's accuracy in detecting missing EB/S2R in a subset of 1,821 bug descriptions. The evaluation indicates that DeMIBuD, with its best setting, detects missing EB (S2R) with 85.9% (69.2%) average precision and 93.2% (83%) average recall. DeMIBuD can be used either to alert submitters while writing bug reports or as a quality assessment tool for triagers, so that they can contact the reporters right away to solicit the missing information while the facts are still fresh in memory. DeMIBuD can also be used to augment existing bug report quality models [33, 78].

In summary, the major contributions of our research are: (1) a set of 154 patterns that capture the discourse followed by reporters when describing OB, EB, and S2R in bug reports; (2) an automated approach (DeMIBuD) to detect the absence/presence of EB and S2R in bug reports; (3) a dataset of labeled bug reports that can be used for replication purposes and future research [23].

2 THE DISCOURSE OF BUG DESCRIPTIONS

The first goal of our research is to understand how essential information about bugs is reported. To this end, we identify the discourse patterns that reporters use to describe OB, EB, and S2R in bug reports. *Discourse patterns* are rules that capture the syntax and semantics of the text. Figures 1, 2, and 3 are examples of OB, EB, and S2R discourse patterns, respectively. To address such a goal, we answer the following research questions (RQs):

RQ$_1$: *To what extent bug reports contain OB, EB, and S2R?* This RQ investigates if reporters tend to include OB, EB, and S2R. This motivates the need for automated detection of such information.

RQ$_2$: *Do bug reporters describe OB, EB, and S2R in bug descriptions by using a well-defined set of discourse patterns?* This RQ aims at understanding *the discourse* followed by reporters to describe OB, EB, and S2R. The presence of discourse patterns is essential to automatically identify such contents.

To answer these questions, we performed a qualitative discourse analysis [12, 57] of a large set of bug reports from nine software projects, based on open coding [49]. Before describing the coding process, the coding criteria, and the coding results, we introduce a set of assumptions and definitions useful for our study.

2.1 Definitions

We focus on *bug reports*, *i.e.*, issues that describe potential software bugs or defects. We do not code issues describing *feature requests*, *enhancements*, *questions*, or *tasks*. In addition, our pattern discovery task focuses on the description of bug reports (*i.e.*, *bug descriptions*) and not on titles. The reason for this is that titles rarely describe completely OB, EB, or S2R, *e.g.*, they can simply be noun phrases or words referring to the reports' topics [39]. We focus our attention on three *types of information* in bug descriptions, namely, *Observed Behavior* (OB), *Expected Behavior* (EB), and *Steps to Reproduce* (S2R). We expect to find a set of *discourse patterns* for the sentences and paragraphs (*i.e.*, the *units of discourse*) of the bug descriptions. A *discourse pattern* is a rule that structures a sentence or a paragraph to convey either OB, EB, or S2R. This means that a pattern captures the syntax and semantics of sentences and paragraphs.

2.2 Issue Sampling

We collected a sample set of issues from nine software projects of different types and domains. These projects rely on different issue (or bug) trackers to capture potential software bugs found by users. *Eclipse* [11], *Firefox* [10], *Httpd* [3], and *LibreOffice* [9] use Bugzilla as issue tracker; *Hibernate* [2] and *OpenMRS* [4] use Jira; *Docker* [7] and *Wordpress-Android* (*a.k.a.* Wordpress-A) [5] use GitHub's Issues; and *Facebook* [8] uses a proprietary issue tracking system. These projects, except for Facebook, are open source.

To create our issue sample set for the coding task, we rely on the issue data set collected by Davies *et al.* [26] for Eclipse, Facebook, Firefox, and Httpd. This data set is composed of 1.6k issues randomly sampled from their corresponding issue trackers. From this data set and the online issue repositories of the remaining projects, we performed random sampling, making sure to exclude issues that were not bug reports (*e.g.*, feature requests) by manually inspecting the type of issue and its comments. In total, we collected 2,912 bug reports, *i.e.*, 324 reports per project on average (including the ones collected by Davies *et al.* [26]). From these, we used 1,091 reports for discourse pattern discovery and the remaining ones (*i.e.*, 1,821) for validation purposes. We refer to the former data set as the *discourse bug reports* and to the latter as the *validation bug reports*.

2.3 Coding Procedure

We present the coding procedure that we followed to address both **RQ$_1$** and **RQ$_2$**. While we coded the presence of OB, EB, and S2R in the *discourse bug reports* and *validation bug reports*, we only used the *discourse bug reports* to infer the discourse patterns.

2.3.1 ***Discourse Pattern Coding.*** Five coders (four authors of this paper and one additional coder) conducted the sentence and paragraph coding task for the *discourse bug reports*. In order to define a starting coding framework, one of the coders conducted a

pilot study on a subset of 25 issues from Davies *et al.*'s issues [26] that were not used in *discourse bug reports*. The goal of this task was to analyze the issue descriptions, identify the sentences or paragraphs that corresponded to OB, EB, and S2R, and infer the discourse patterns from them. This task resulted in twelve preliminary discourse patterns with specific textual examples from the issues, a set of textual characteristics of the issues, and the initial coding criteria. Once the pilot study was completed, this person trained the rest of the coders in a 45-minute session that involved discussing the results and some ambiguous sentences.

The 1,091 *discourse bug reports* were evenly and randomly distributed among coders, to ensure that each coder received a subset of reports for each of the nine projects. Each person coded 218 reports except for one person who coded 219 (*i.e.*, 25 reports per system per person, on average). For each bug report, the coders analyzed the bug description and marked each sentence or paragraph as OB, EB, or S2R. A sentence/paragraph can fall into more than one of these categories at the same time. Then, the coders inferred a discourse pattern from each marked sentence/paragraph and assigned a *code* to it. A *code* is a label that uniquely identifies a discourse pattern. Note that it is possible to infer more than one pattern from a sentence/paragraph. A catalog of inferred patterns was shared among coders via an online spreadsheet. In this way, all coders were aware of the patterns inferred by each coder and were able to reuse existing patterns or add new ones to the catalog. When one of the coders identified a new pattern, it was included in the catalog and the other coders were notified. Each new pattern was verified by all the coders and disagreements were solved via open discussion. For each new pattern, the existing catalog was inspected for similar patterns and, when appropriate, with unanimous agreement, similar patterns were merged into a new one (*i.e.*, a more general pattern) and the existing labels were updated accordingly. This process was fully iterative, and included constant refinement of the pattern catalog as well as discussion of ambiguous cases. Every decision taken during the pattern extraction was representative of the opinion of all coders.

To minimize subjectivity, we recruited four additional coders (one CS masters student, two developers, and one business analyst) and asked them to code the same 1,091 reports coded by the first group of coders. In a 40-minute session, one member of the first group trained the new coders on the coding procedure and criteria (see Section 2.3.3). We randomly distributed the reports among the new coders ensuring that each one coded a subset of issues coded by each of the original coders. The task of the additional coders was to mark the sentences and paragraphs that corresponded to OB, EB, and S2R. This time, the pattern inference was not part of the task, as the iterative and collaborative nature of the pattern coding procedure already aimed at minimizing subjectivity. In the end, each issue from the *discourse bug reports* was coded by two distinct coders. The inter-coder agreement is discussed in Section 2.4.1.

2.3.2 ***Validation Set coding.*** For the remaining 1,821 bug reports from our initial sample, *i.e.*, the *validation bug reports*, all nine coders were requested to follow the same coding process, without pattern inference. Each report was coded by two different coders, *i.e.*, on average 202 issues were assigned to each pair of coders. The bugs were randomly distributed so that each pair of coders received a subset of issues from each system. Again, the coders marked the sentences and paragraphs that corresponded to OB, EB, and S2R (*i.e.*, no pattern inference this time).

2.3.3 ***Coding Criteria.*** We summarize the most important criteria followed by the coders (full list in our replication package [23]). The coders were provided with examples of each criterion.

The coding focused only on natural language (NL) content written by the reporters, as opposed to code snippets, stack traces, or logs. However, the NL referencing this information was coded. In addition, only explicit mentions of OB/EB/S2R were labeled. Note that it is possible to infer EB from OB descriptions, as the former is usually the opposite of the latter. Such cases were not labeled.

Regarding OB, uninformative sentences such as *"The system does not work"* are insufficient to be considered OB. There must be a clear description of the observed (mis)behavior of the software. Code explanations and root causes are not considered OB. Regarding EB, solutions or recommendations to solve the bugs are not considered EB. In some cases, imperative sentences such as *"Make Targets not automatically filled..."* may be considered EB according to the context of the bug. Sometimes, however, these suggest tasks instead of EB. Regarding S2R, one or more sentences (*i.e.*, a sentence or a paragraph) can describe steps to reproduce. Conditional sentences such as *"when I click on apache.exe it returns an error like this"* may be S2R, if they provide enough details about how to reproduce the bug. Finally, S2R paragraphs may also contain OB and EB sentences.

2.4 Coding Results and Analysis

Before reporting and discussing the coding results, we briefly summarize the inter-coder agreement measurements.

2.4.1 ***Inter-coder Agreement.*** We analyzed the reliability of the coding process regarding the presence and absence of OB, EB, and S2R in bug descriptions. Remember that each bug description was coded by two coders. We measured the observed agreement between coders as well as Cohen's Kappa (k) [25] and Krippendorff's alpha (α) [40] coefficients. Our analysis reveals high inter-coder agreement levels. Coders agreed on: the presence or absence of OB in 91% of the cases (avg. k = 37.3%, α = 40.4%, *i.e.*, fair agreement [70]); the presence or absence of EB in 85.5% of the cases (avg. k = 70.2%, α = 67.7%, *i.e.*, substantial agreement [70]); and the presence or absence of S2R in 76% of the cases (avg. k = 49.2%, α = 51.9%, *i.e.*, moderate agreement [70]).

Overall, 1,131 bug reports (*i.e.*, 38.9%) had some type of disagreement, solved by applying a third person scheme. We distributed the conflicting reports among the nine coders in such a way that a third coder (different from the original two coders) would judge and solve the disagreements. Our analysis revealed that the main causes for disagreement were omissions, mistakes, and, in the case of S2R, misunderstandings, as in several cases it was not clear if (single) conditional sentences were S2R or not.

2.4.2 ***RQ_1: Presence of OB, EB, and S2R in Bug Reports.*** Table 1 reveals that, while most of the bug reports contain OB (*i.e.*, 93.5%), only 35.2% and 51.4% of the reports explicitly describe EB and S2R, respectively. 22.1% of the reports contain all three types of information (OB, EB, and S2R). These results indicate that essential information is missing in bug reports and motivate the need for

automated detection of such information. Firefox is the system with the highest number of reports having EB and S2R (*i.e.*, 67.4% and 76.4%) and having all three types of information. We attribute this result to the use of predefined templates explicitly asking for this information. Wordpress-Android has the lowest number of reports with OB. We observed that screenshots, rather than textual descriptions, are commonly used in this project.

Table 1: Number of bug reports containing OB, EB, and S2R.

Project	#OB	#EB	#S2R	Total
Docker	314 (93.5%)	113 (33.6%)	207 (61.6%)	**336**
Eclipse	271 (90.9%)	101 (33.9%)	173 (58.1%)	**298**
Facebook	327 (96.7%)	81 (24.0%)	133 (39.3%)	**338**
Firefox	335 (96.5%)	234 (67.4%)	265 (76.4%)	**347**
Hibernate	315 (95.2%)	89 (26.9%)	150 (45.3%)	**331**
Httpd	350 (96.4%)	102 (28.1%)	104 (28.7%)	**363**
LibreOffice	322 (97.3%)	122 (36.9%)	241 (72.8%)	**331**
OpenMRS	275 (93.5%)	94 (32.0%)	104 (35.4%)	**294**
Wordpress-A	215 (78.5%)	88 (32.1%)	121 (44.2%)	**274**
Total	**2,724 (93.5%)**	**1,024 (35.2%)**	**1,498 (51.4%)**	**2,912**

2.4.3 ***RQ$_2$: Bug Descriptions Discourse***. Our open coding approach resulted in a catalog of 154 patterns that capture the discourse followed by reporters to describe OB, EB, and S2R. Most of the patterns are sentence-level patterns (135) and most of the paragraph-level patterns correspond to S2R (13 out of 19). We summarize and discuss our pattern catalog and the discourse used for each type of information in bug descriptions (the full catalog is available in our replication package [23]).

OB discourse. We observe that many patterns in our catalog correspond to OB (*i.e.*, 90 or 58.4% — see Table 2). Out of these, 85 are sentence-level patterns and five are paragraph-level patterns. Software (mis)behavior is usually described following a negative discourse. The six most frequent OB patterns correspond to negative textual content and account for 68.9% of the *discourse bug reports* that contain OB. Three of these patterns are: NEG_AUX_VERB, VERB_ERROR, and NEG_VERB. The first one is the most frequent one, which corresponds to negative sentences containing auxiliary verbs (see Fig. 1). The second one corresponds to sentences with verb phrases containing error-related nouns, such as *"VirtualBOx GUI gives this error:"* (from Docker 1583), and the third one, to sentences with non-auxiliary negative verbs such as *"Writer hangs on opening some doc, docx or rtf files"* (from LibreOffice 55917). We also observed OB positive discourse. For instance, the COND_POS pattern represents conditional sentences with positive predicates, such as *"When the merge was completed, I saw that the entries in the value_coded column remained as they were originally"* (from OpenMRS TRUNK-3905). The BUT pattern corresponds to sentences containing contrasting terms followed by affirmative predicates, such as *"You require at least 7 letters, but our name (Delupe) only consists of 6"* (from Facebook 13084). The top six most frequent positive discourse patterns account for 33% of the reports describing OB. Overall, the top six most frequent negative and the top six positive patterns appear in 82.5% of the OB bug descriptions.

EB discourse. Reporters describe expected behavior using 31 patterns (*i.e.*, 20.1% of our pattern catalog—see Table 2). Most of them (*i.e.*, 30) are sentence-level patterns. The most frequent pattern is SHOULD (see Fig. 2), which represents sentences using the modal terms *"should"* or *"shall"*. These types of sentences appear in 44.2% of the reports that describe EB. Other frequent discourse for describing EB is represented by the EXP_BEHAVIOR, INSTEAD_OF_EXP_BEHAVIOR, EXPECTED, and WOULD_BE patterns. The former, EXP_BEHAVIOR, represents sentences with explicit EB labels, such as *"Expected Results: Taken away the dialog box..."* (from Firefox 226732); INSTEAD_OF_EXP_BEHAVIOR accounts for sentences using *"instead of"* (or similar terms), such as *"When you try to schedule a saved draft, it is published immediately instead of being scheduled for the future date you select"* (from Wordpress-Android 3913); the EXPECTED pattern represents sentences using noun phrases or conjugated verbs of the word *"expect"*, such as *"The expectation was that objects would be loaded identically regardless of using scrollable results or using get result list from JPA."* (from Hibernate HHH-10062); and WOULD_BE corresponds to sentences containing *"would be + positive adjective"* phrases, such as *"It'd be optimal if the UX updated to reflect the actual updated follow state for given users/blogs"* (from Wordpress-Android 447). These five patterns appear in 86.3% of the reports describing EB.

Pattern code: S_OB_NEG_AUX_VERB
Description: negative sentence with auxiliary verbs
Rule: ([subject]) [negative aux. verb] [verb] [complement]
Definitions:
[negative aux. verb] ∈ {are not, can not, does not, did not, *etc.*}
Example: [*The icon*] **[*did not*]** [*change*] [*to an hourglass...*] (from Eclipse 150)

Figure 1: Most common OB discourse pattern.

Pattern code: S_EB_SHOULD
Description: sentence using the modals "should" or "shall" with no preceding predicates that use negative auxiliary verbs
Rule: [subject] should/shall (not) [complement]
Example: [*Apache*] ***should*** [*make an attempt to print the date in the language requested by the client*] (from Httpd 40431)

Figure 2: Most common EB discourse pattern.

S2R discourse. The steps to reproduce discourse is represented by 33 patterns (see Table 2), 13 of which are paragraph-level patterns. This means that reporters often use more than one sentence to describe steps to reproduce. While the most frequent pattern to describe S2R is paragraphs containing a labeled list of actions (see Fig. 3—*i.e.*, it accounts for 30.7% of the reports describing S2R), S2R is also expressed using a single sentence. For example, the COND_OBS pattern corresponds to conditional sentences containing non-negative OB predicates, such as *"When saving a new (transient) entity ..., Hibernate will generate [at least] two INSERT statements..."* (From Hibernate HHH-6630). In addition, the CODE_REF pattern describes sentences with noun phrases and adverbs of location to refer to code, scripts, or other non-natural language information used to reproduce the observed behavior. An example of this type of sentences is: *"The following statement produces a compilation error in JDT..."* (from Eclipse 52363). The top five most frequent S2R discourse patterns are present in 77.2% of the S2R bug descriptions.

Unique discourse patterns. We found overlap among OB, EB, and S2R patterns. Either some patterns are equivalent or they are part of others across the three types of information. Specifically, we found that the INSTEAD_OF OB pattern is equivalent to the

Pattern code: P_SR_LABELED_LIST
Description: paragraph containing a non-empty labeled list of sentences that indicate actions. The label is optional and indicates S2R terms. The "action sentences" may be simple or continuous present/past sentences or imperative sentences. The list may contain OB and EB sentences in no particular order.
Rule: ([S2R label])
[[number/bullet] [action sentence]]+
[([number/bullet]) [OB/EB sentence]]*
Definitions:
- [S2R label] ∈ {"how to reproduce", "STR", "To replicate", "Steps to reproduce", ... }
- [number/bullet] ∈ { "1.", "1 -", "-", "*", ... }
- [action sentence] ∈ {[present/past continuous sentence], [simple present/past sentence], [imperative sentence]}

Example: (from Firefox 215939)
[*Steps to Reproduce:*]
[*1.*] [*Start Firebird.*]
[*2.*] [*C-t to open a new tab.*] [*The second tab is now displayed.*]
[*3.*] [*Type 'hello'.*] [*This text appears in the location bar.*]
[*4.*] [*Click on the header for the first tab to switch to that tab.*]
[*5.*] [*Click on the header for the second tab...*]

Figure 3: Most common S2R discourse pattern.

INSTEAD_OF_EXP_BEHAVIOR EB pattern. The COND_POS and COND_NEG OB patterns are part of two S2R paragraph-level patterns and three S2R sentence-level patterns (all related to conditional content). The IMPERATIVE EB pattern is part of two S2R paragraph-level and two S2R sentence-level patterns (all related to imperative content). In the end, 87 OB, 29 EB, and 24 S2R patterns are unique in our catalog (see our replication package [23]).

Summary. Our discourse analysis revealed that reporters use 154 discourse patterns to describe OB, EB, and S2R; and 82% (on average) of the instances of OB, EB, and S2R are described using only 22 (14.3%) of the patterns.

2.4.4 ***Discourse Analysis across Projects.*** Table 2 shows how many patterns of every kind we identified in any of the sentences/paragraphs of the bug reports for each project. As seen in the table, not all patterns are used in each system.

Table 2: Number of patterns used to express OB, EB, and S2R.

Project	# of reports	# of patterns			
		Overall	OB	EB	S2R
Docker	113	88	64	11	13
Eclipse	148	95	64	12	19
Facebook	153	102	67	16	19
Firefox	132	97	62	17	18
Hibernate	103	86	64	10	12
Httpd	133	100	72	13	15
LibreOffice	120	88	59	10	19
OpenMRS	92	70	48	11	11
Wordpress-A	91	69	44	11	14
Overall	**1,085**	**154**	**90**	**31**	**33**

Fig. 4 depicts the pattern distribution across projects, *i.e.*, each bar indicates the number of patterns that appear in a given number of projects. Out of the 154 patterns in our catalog, 21 (*i.e.*, 13.6%) patterns appear in only one project each, 42 (*i.e.*, 27.3%) appear in two to four projects, 67 (*i.e.*, 43.5%) appear in five to eight projects, and 24 patterns (*i.e.*, 15.6%) appear in all nine projects. Examples of rare patterns that appear in one project only are: IMPOSSIBLE, which is used to describe OB in *"...it's impossible to click the post button..."* (from Facebook 8978); WHY_FIRST_PLACE, which is used to describe EB in *"...why the outgoing changes didn't appear in the first place"* (from Eclipse 41883); and COND_THEN_SEQ, which is used to describe S2R in *"If you enter..., then switch..., then switch back... "* (from Firefox 215939). Examples of frequently used patterns which appear in all nine projects are: NEG_AUX_VERB (OB), SHOULD (EB), and COND_OBS (S2R)—these are previously described in Section 2.4.3.

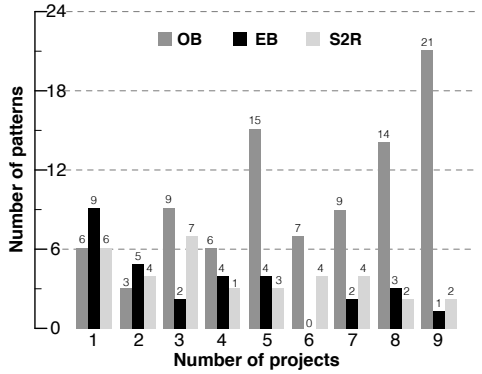

Figure 4: Distribution of pattern appearance across projects.

Fig. 4 reveals that the distribution of OB patterns differs from the distribution of EB and S2R patterns. Nearly three quarters of the OB patterns (*i.e.*, 66 or 73.3%) appear in more than half of the projects (*i.e.*, in five to nine projects). In contrast, less than a third of the EB patterns (*i.e.*, 10 or 32.3%) and less than half of the S2R patterns (*i.e.*, 15 or 45.5%) appear in more than half of the projects. These results indicate that reporters use many patterns to describe OB and they reuse them often across systems. Reporters use far fewer patterns to describe EB and only a third of them are reused frequently. About half of the S2R patterns are reused frequently across systems.

Our previous analysis identified 22 patterns that are used in 82% of the OB/EB/S2R descriptions, on average. We deepened our analysis to determine how many patterns are used to achieve similar coverage for *each* system in our sample set. Nearly a third of the patterns (*i.e.*, 49 out 154—31.8%) are used to describe OB, EB, or S2R in at least 82% of the cases. Among the 22 patterns, twenty (10 OB, 5 EB, and 5 S2R patterns) are frequently reused in each project.

3 DEMIBUD: DETECTING MISSING INFORMATION IN BUG DESCRIPTIONS

Our first study revealed that, while most of the bug reports (93.5%) contain OB, only 35.2% and 51.4% of the reports explicitly describe EB and S2R, respectively. These results motivate the need for an automatic approach to detect the absence of this information in bug descriptions. We focus on detecting EB and S2R, as OB is described in nearly all bug reports. Our study also indicated that reporters follow a relatively limited set of discourse patterns to describe EB and S2R across systems (*i.e.*, around 30 in each case). The existence of these patterns confirms our original conjecture and supports our research on automatically identifying missing EB and S2R.

We designed and evaluated three versions of DeMIBuD that automatically detect missing EB and S2R in bug descriptions: DeMIBuD-R, based on regular expressions; DeMIBuD-H, based on heuristics and NLP; and DeMIBuD-ML, based on machine learning. The first two approaches are unsupervised, while the third one requires the

use of a labeled set of bug reports explicitly reporting the ones containing or not EB and S2R.

3.1 Regular Expressions-based DeMIBuD

DeMIBuD-R uses regular expressions to detect if a bug report contains (or not) EB and S2R. The regular expressions rely on frequently used words found in our EB and S2R discourse patterns, such as keywords explicitly referring to EB or S2R (*e.g.*, "expected result/behavior" or "steps to reproduce/recreate"), and keywords commonly used to describe EB (*i.e.*, modal verbs such as "should", "could", or "must", or other terms such as "instead of"). For S2R, DeMIBuD-R also detects enumerations (*e.g.*, 1., 2., *etc.*) and itemizations (*e.g.*, '*', '-', *etc.*). If any of the sentences or paragraphs of a bug report matches any of the regular expressions, then DeMIBuD-R labels the report as containing EB/S2R, otherwise, DeMIBuD-R labels it as missing EB/S2R. DeMIBuD-R extends existing approaches to detect EB/S2R [17, 26, 78]. The full list of regular expressions used by DeMIBuD-R can be found in our replication package [23].

3.2 Heuristics-based DeMIBuD

DeMIBuD-H uses part-of-speech (POS) tagging and heuristics to match sentences and paragraphs to our discourse patterns. We implemented each one of the patterns in our catalog by using the Stanford CoreNLP toolkit [47]. For example, to detect EB sentences that follow the discourse pattern SHOULD, DeMIBuD-H first identifies the clauses of a sentence by finding coordinating conjunctions (*i.e.*, tokens tagged as "CC") or punctuation characters (*e.g.*, commas), and splits the sentence using these tokens. Then, for each clause, it identifies the modal terms "should" or "shall" by processing the tokens labeled as "MD" (*i.e.*, modal). Finally, DeMIBuD-H checks for the absence of any predicate that uses negative auxiliary verbs prior to the modal. This is done by identifying the adverb "not" preceded by auxiliary verbs, *i.e.*, the verbs[1] "do", "have", or "be", or the modals "can", "would", "will", "could", or "may". DeMIBuD-H also checks for the complement after the modal and for some exceptions (*e.g.*, phrases, such as "should be done"). If any of the clauses satisfy these rules, then the sentence is detected as following the SHOULD discourse pattern and labeled as an EB sentence. Each pattern implementation is used to classify all sentences/paragraphs in a bug description as having or not having EB/S2R. A bug report is labeled as containing EB/S2R if at least one sentence/paragraph of the bug report matches any EB/S2R pattern implementation. Otherwise, the bug report is labeled as missing EB/S2R.

3.3 Machine Learning-based DeMIBuD

DeMIBuD-ML is based on state-of-the-art approaches in automated discourse analysis and text classification [20, 21, 35], which utilize textual features, such as *n-grams* and *POS tags* (*i.e.*, part of speech tags) [35]. DeMIBuD-ML relies on two binary classifiers, one that detects missing EB, and another one that detects missing S2R.

Textual Features. We use our *discourse patterns* as features of bug descriptions for classification purposes. Our patterns capture the structure and (to some extent) the vocabulary of the descriptions. Each EB and S2R pattern is defined as a boolean feature indicating the presence or absence of the pattern in any of the sentences/paragraphs of a bug report. We use the pattern implementations of DeMIBuD-H to produce the pattern features. EB and S2R features are used in turn by the corresponding classifier (*i.e.*, the one for EB or S2R, respectively). We also use *n-grams* to capture the vocabulary of bug descriptions. *N*-grams are contiguous sequences of *n* terms in the text. We use unigrams, bigrams, and trigrams, where each *n*-gram is defined as a boolean feature indicating the presence or absence of such an *n*-gram in any of the sentences of a bug report. Finally, we use *POS tags* to capture the type of vocabulary used in the bug descriptions. Similar to *n*-grams, we use contiguous sequences of *n*-POS tags in the text. We define {1, 2, 3}-POS tags as boolean features indicating the presence or absence of a tag combination in any of the sentences of a bug report.

Learning Model. Our current implementation of DeMIBuD-ML uses linear Support Vector Machines (SVMs) (from SVM-Light [34]) to classify the bug reports as missing or not missing EB and S2R. Linear SVMs are robust state-of-the-art learning algorithms for high-dimensional and sparse data sets, commonly used for text classification based on *n*-grams [34–36]. Investigating the use of other classifiers is subject of future work.

[1] A verb is a token labeled with one of the VB*x* POS tags, such as VBD or VBN (*i.e.*, verb in past tense or past participle).

Table 3: Bug reports missing EB/S2R (*validation bug reports*).

Project	# missing EB	# missing S2R	Total
Docker	145 (65.3%)	82 (36.9%)	**222**
Eclipse	98 (66.2%)	63 (42.6%)	**148**
Facebook	137 (74.9%)	113 (61.7%)	**183**
Firefox	78 (36.3%)	56 (26.0%)	**215**
Hibernate	169 (74.1%)	118 (51.8%)	**228**
Httpd	172 (74.8%)	173 (75.2%)	**230**
LibreOffice	131 (62.1%)	57 (27.0%)	**211**
OpenMRS	135 (67.2%)	140 (69.7%)	**201**
Wordpress-A	126 (68.9%)	108 (59.0%)	**183**
Total	**1,191 (65.4%)**	**910 (50.0%)**	**1,821**

3.4 Empirical Evaluation Design

We conducted an empirical evaluation with the *goal* of determining how accurately DeMIBuD can detect missing EB and S2R in bug descriptions and comparing the accuracy of the different instances of DeMIBuD. The *context* of our study is represented by the *validation bug reports* from the nine software projects used for open coding. This data set is our gold set (see Table 3). The empirical evaluation aims to answer the following research question:

RQ$_3$: *Which DeMIBuD strategy has the highest accuracy in detecting missing EB and S2R content in bug descriptions?*

We describe the methodology we used to answer **RQ$_3$**, *i.e.*, text preprocessing, approach tuning, evaluation settings, and metrics.

Text Preprocessing. We removed uninformative text that is likely to introduce noise to the detection using different text preprocessing strategies. Specifically, we performed *code removal*, *i.e.*, deletion of code snippets, stack traces, output logs, environment information, *etc.* This was done by using regular expressions and heuristics, defined after our observations of the text. We also performed *basic preprocessing*, *i.e.*, replacing URLs with the "_URL_" meta-token, and removing special characters (*e.g.*, punctuation), numbers, single characters, and tokens starting with numbers. In addition, we performed *stemming* [6], and *stop-word removal*, *i.e.*,

deletion of common articles, prepositions, or adjectives, by using an adapted version of the Lemur stop word list [54]. Our replication package contains the preprocessed bug descriptions, the list of stop words, and the code removal implementation [23].

When assessing the performance of DeMIBuD-R and DeMIBuD-H, we only use *code removal* as the preprocessing strategy, since, by design, these approaches need special characters (*e.g.*, the ones used for itemizations and enumerations), unstemmed vocabulary, and stop words (*e.g.*, "if", "when", "then", *etc.*). The performance of DeMIBuD-ML is determined by using the combination of all preprocessing strategies mentioned above.

Tuning and Evaluation Settings. We used the *discourse bug reports* to test DeMIBuD-R and DeMIBuD-H. This data set contains positive and negative instances that allowed us to test and tune our implementations. Since DeMIBuD detects the absence of EB/S2R, a *negative instance* is a sentence/paragraph/report that contains an explicit description of EB/S2R, whereas a *positive instance* is a sentence/paragraph/report that misses such a description.

By using the *discourse bug reports*, we determined the patterns that contribute most (and least) to DeMIBuD-H's accuracy. We followed a leave-one-out strategy for each one of the EB and S2R patterns. Having all the patterns activated, we deactivated one pattern at a time and measured DeMIBuD-H's accuracy (*i.e.*, the F_1 score—more details below) without using such pattern. Overall, we identified three patterns that, when deactivated, drastically deteriorate the accuracy of DeMIBuD-H (*i.e.*, the F_1 score drops drastically[2]). These patterns are: SHOULD for EB; and LABELED_LIST and AFTER for S2R. Conversely, we also identified three patterns that drastically improve the accuracy of DeMIBuD-H when they are deactivated, namely, CAN and IMPERATIVE for EB, and CODE_REF for S2R. These latter three patterns negatively affect DeMIBuD-H's performance because they occur frequently in sentences that do not describe EB/S2R. Hence, we call these as "ambiguous patterns". We measured DeMIBuD-H's accuracy both by using all the patterns and by omitting the ambiguous patterns.

For DeMIBuD-ML, we performed 10-fold cross validation (10CV) using the *validation bug reports*. To avoid over-fitting [29], we used 70%, 20%, and 10% of the bug reports for training, parameter tuning, and testing, respectively. This strategy ensures that all bug reports are used for training, parameter tuning, and testing. The testing data set was used to measure DeMIBuD-ML's accuracy. To follow a realistic approach, we performed 10CV independently on the bug reports of each project. We call this setting *within-project evaluation*. We used stratified sampling to create the folds, thus ensuring that the proportions of negative and positive instances are similar to the proportions of all the reports in the corresponding project (remember that a negative instance indicates the presence of EB/S2R, while a positive instance indicates the absence). To assess feature generality in DeMIBuD-ML, we also conducted a *cross-project evaluation*, in which the bug reports of one project were used for testing, and the reports of the remaining eight projects were used for training and parameter tuning (approximately 80% and 20% of the reports were used for training and parameter tuning, respectively).

In our experiments, we tuned the penalty parameter C of the linear SVMs by using the parameter tuning data set of each fold. Larger C values mean higher penalty on errors. We experimented with the following parameter values: 1×10^{-4}, 2.5×10^{-4}, 5×10^{-4}, 7.5×10^{-4}, ..., 5, 7.5. We chose the best parameter C by maximizing the F_1 score of the trained SVMs to detect missing EB and S2R. We found that the parameters that lead to the best accuracy fall in the ranges [0.05, 0.5] and [0.0025, 0.1] for EB and S2R, respectively.

Evaluation Metrics. We use standard metrics in automated classification to measure the accuracy of our approaches, namely, precision, recall, and F_1 score [29]. Precision is the percentage of bug reports predicted as missing EB/S2R that are correct with respect to the gold set (*i.e.*, $Precision = TP/(TP + FP)$). Recall is the percentage of bug reports missing EB/S2R that are correctly predicted as missing EB/S2R (*i.e.*, $Recall = TP/(TP + FN)$). F_1 score is the harmonic mean of precision and recall, which gives a combined measure of accuracy.

Intuitively, we prefer higher recall, as in a practical setting, we want DeMIBuD to alert reporters whenever EB or S2R is missing in their bug descriptions. Nonetheless, we also want DeMIBuD to achieve high precision, as many false alerts would hinder its usability. Experiments with users are needed to assess acceptable trade-offs between recall and precision. We leave such studies for future work. In this paper, we focus on the F_1 score as an accuracy indicator, as we did for the tuning. When two configurations yield the same F_1 score, we prefer the one with higher recall.

3.5 Results and Discussion

We present and discuss the accuracy achieved by our three instances of DeMIBuD when detecting the absence of EB and S2R using different strategies and features (see Table 4).

[2]For space reasons, we omit the results of this tuning approach. However, they can be found in our replication package [23].

Table 4: Overall *within-project* detection accuracy of the different instances of DeMIBuD.

Approach	Strategy or Features	EB			S2R		
		Avg. Prec.	Avg. Recall	Avg. F_1	Avg. Prec.	Avg. Recall	Avg. F_1
DeMIBuD-R	-	86.0%	85.9%	85.9%	63.3%	92.4%	74.3%
DeMIBuD-H	all patterns	96.7%	46.1%	62.2%	84.5%	31.0%	44.3%
DeMIBuD-H	no ambiguous patterns	95.1%	76.6%	84.7%	81.6%	38.5%	51.2%
DeMIBuD-ML	pos	73.8%	93.1%	82.0%	60.8%	75.8%	66.8%
DeMIBuD-ML	*n*-gram	75.1%	97.6%	84.7%	66.4%	83.4%	73.4%
DeMIBuD-ML	pos + *n*-gram	76.0%	95.1%	84.2%	65.3%	79.2%	71.1%
DeMIBuD-ML	patterns	85.9%	93.2%	89.4%	63.5%	80.3%	70.7%
DeMIBuD-ML	patterns + pos	77.9%	92.9%	84.6%	65.4%	76.0%	69.9%
DeMIBuD-ML	patterns + *n*-gram	76.9%	97.0%	85.6%	69.2%	83.0%	74.9%
DeMIBuD-ML	pos + patterns + *n*-gram	76.8%	95.8%	85.1%	67.2%	80.9%	73.0%

3.5.1 ***DeMIBuD-R's Accuracy.*** DeMIBuD-R achieves 85.9% avg. recall and 86% avg. precision when detecting missing EB (see Table 4). Based on F_1, this is the second best approach across all versions of DeMIBuD. Our analysis reveals that DeMIBuD-R fails to detect missing EB in 168 bug reports that do not describe EB (*i.e.*, false negatives). This is mainly due to the inherent imprecision of keyword matching via regular expressions. For example, we found usages of modal verbs to express OB instead of EB, as in *"This problem could also be related to some sites not copying URLs..."* (from Firefox 319364), and modal verbs appearing in error messages, *e.g.*, *"... and the error could not load the item appeared on the screen"* (from Wordpress-Android 859). We also observed that DeMIBuD-R detects missing EB in 167 bug reports that describe EB (*i.e.*, false positives), as they do not match the keywords used by DeMIBuD-R. For example, we found EB sentences phrased with *"used to"*, as in *"...you used to be able to add new Obs to an already existing encounter order..."* (from OpenMRS TRUNK-211).

Regarding S2R, DeMIBuD-R achieves the highest recall (93.4%) but also one of the lowest precision values (*i.e.*, 63.3%) across the different versions of DeMIBuD. DeMIBuD-R's recall suggests that bug reports missing S2R usually do not contain explicit S2R keywords and/or itemizations/enumerations. Yet, in the few false negatives produced by DeMIBuD-R (*i.e.*, 18), we found non-S2R sentences using S2R keywords, such as *"I tried to reproduce the issue without luck..."* (from Wordpress-Android 1318), or templates that contain S2R keywords but are filled in with non-S2R content, *e.g.*, *"Steps To Reproduce: Unsure how to reproduce..."* (Eclipse 229806). DeMIBuD-R flagged missing S2R in 530 bug reports describing S2R (*i.e.*, false positives). This is somehow expected as users describe S2R using alternative wordings to enumerations/itemizations, which are not keyword specific. For example, users can describe S2R in a narrative way: *"Open the history view on a file with interesting revisions. Click the date column to sort by date..."* (from Eclipse 17774). Overall, DeMIBuD-R ranked as the second most accurate detector across all versions of DeMIBuD, in terms of F_1 score (*i.e.*, 74.3%). The results indicate that DeMIBuD-R is accurate in detecting missing S2R, yet produces a rather large number of false alarms.

3.5.2 ***DeMIBuD-H's Accuracy.*** When all the patterns are used, DeMIBuD-H is able to detect missing EB with 46.1% recall and 96.7% precision. When we deactivate the ambiguous EB patterns (*i.e.*, IMPERATIVE and CAN), DeMIBuD-H's recall improves substantially (*i.e.*, from 46.1% to 76.6%) at almost the same precision (*i.e.*, 95.1%). This large recall improvement is explained by the large number of bug reports missing EB that contain sentences matching the ambiguous patterns, which lead to many false negatives (*i.e.*, failing to detect missing EB). We found 438 and 305 reports missing EB that contain IMPERATIVE and CAN sentences (*i.e.*, 36.8% and 25.6% of the bug reports that do not describe EB), respectively. We observe that IMPERATIVE sentences are usually used to describe S2R, *e.g.*, *"1. Create a container with volumes in docker 1.8.3"* (from Docker 18467), or ask for information to the reporter via templates, *e.g.*, *"**Describe the results you received:**"* (from Docker 27112). CAN sentences describe other non-EB content, *e.g.*, *"the user can only tell the difference when he recognizes..."* (from Firefox 293527).

When the ambiguous EB patterns are deactivated, we observe that the main reason for false negatives is sentences describing non-EB content, yet following the SHOULD EB pattern. We found conditional sentences expressing actions, *e.g.*, *"If that's the case, we should document this on the wiki..."* (from OpenMRS TRUNK-4907); questions using the modal "should", *e.g.*, *"... should following tags be unavailable while signed out?"* (from Wordpress-Android 3270); and sentences expressing other type of non-EB content, *e.g.*, *"OpenMRS shouldn't bomb in this situation"* (from OpenMRS TRUNK-2992). Our analysis of the 50 false positives produced by DeMIBuD-H revealed that our pattern implementation is unable to match some sentences. In addition, we found a handful of bug reports containing EB sentences that are not captured by any of our EB patterns, *e.g.*, *"...works as expected (as in the process is not killed)"* (from Docker 11503), or *"With FF2, the user sees the tab transition smoothly to the new tab with no nasty white flash"* (from Firefox 393335).

Regarding S2R, when all the patterns are used, DeMIBuD-H has the lowest recall (*i.e.*, 31%) but the highest precision (*i.e.*, 84.5%). We observe 7.5% recall improvement and 2.9% precision deterioration when DeMIBuD-H relies on all the patterns except CODE_REF (*i.e.*, the ambiguous S2R pattern). We found 338 bug reports missing S2R but containing sentences matching the CODE_REF pattern (*i.e.*, 37.1%). The main reasons behind the false negatives are the imprecision of our implementation (*i.e.*, regarding heuristics, sentence parsing, or code preprocessing) and the presence of ambiguous sentences, such as *"Here are the definitions of the file systems:"* (from Httpd 37077). When the CODE_REF pattern is deactivated, we observe two main reasons for false negatives, namely, the imprecision of our implementation and ambiguous content (*i.e.*, sentences and paragraphs describing non-S2R content yet following other S2R patterns). Regarding the latter, we found non-S2R paragraphs and sentences phrased imperatively that describe solutions, *e.g.*, *"Possible solutions: ... 1. Disable Tomcat's default ..."* (from OpenMRS TRUNK-1581); or actions that do not intend to replicate the OB, *e.g.*, *"See the user edit page for how the void patient..."* (from OpenMRS TRUNK-1781). Other ambiguous cases include non-S2R conditional sentences describing high-level tasks, *e.g.*, *"I noticed that HSQLDB is not enforcing... while trying to troubleshoot a particular..."* (from OpenMRS TRUNK-27); and sentences that convey actions expressed in present perfect tense, present tense, or past tense, *e.g.*, *"I also asked about this in the Hibernatate... and Steve Ebersole said that..."* (from OpenMRS TRUNK-2). These types of discourse are also used to describe S2R and are captured by our S2R patterns. Our analysis of false positives produced by DeMIBuD-H revealed content ambiguity and unusual text structure as the main reasons for hindering precision. We found labeled lists of S2R where each step was written as a separate paragraph (as in LibreOffice 77431); paragraphs containing different sentences that describe S2R and other types of information, *e.g.*, *"I'm using LibreOffice 4.3.6.2... I downloaded 4.3.7 and installed... And I knew 4.3.7 requires..."* (from LibreOffice 91028); itemizations describing OB rather than S2R (as in LibreOffice 78202); sentences not related to OB replication, *e.g.*, *"I am seeing junk characters and I have to change the encoding setting manually"* (from Httpd 49387); and ambiguous sentences describing actions *"I fixed the problem by using..."* (from Httpd 42731). Overall, we observed more content ambiguity related to S2R than to EB. This is one of the reasons for the lower accuracy of DeMIBuD-H (and other DeMIBuD versions) when detecting missing S2R.

DeMIBuD-H's high precision and low recall (when detecting missing EB and S2R) are explained by the focus of our pattern implementations on identifying all different ways to describe EB and S2R (*i.e.*, identifying EB/S2R in most bug reports), without focusing on filtering non-EB/S2R content that is similar to EB/S2R (*i.e.*, it incorrectly predicts EB/S2R in many bug reports). Compared to DeMIBuD-R, DeMIBuD-H's overall accuracy is lower when detecting missing EB and S2R in bug descriptions (in terms of F_1 score). The two main reasons for such (in)accuracy are: (1) imprecision of our heuristics, and (2) ambiguous content in bug descriptions. While the former issue may be addressed by refining some of the patterns, the latter one is more challenging. In any case, DeMIBuD-H's main advantage over the other versions of DeMIBuD is its ability to produce very few false alarms.

3.5.3 ***DeMIBuD-ML's Accuracy.*** When detecting missing EB, DeMIBuD-ML achieves the highest recall (*i.e.*, between 92.9% and 97.6%) at the expense of precision (*i.e.*, between 73.8% and 85.9%). The features used by DeMIBuD-ML that lead to the highest (*i.e.*, 97.6%) and lowest (*i.e.*, 92.9%) recall are *n-grams* and *patterns + POS tags*, respectively. The features that lead to the highest (*i.e.*, 85.9%) and lowest (*i.e.*, 73.8%) precision are *patterns* and *POS tags*, respectively. We observe that *n-grams* always increase recall when combined with other features, and *POS tags* deteriorate recall when combined with *patterns*. *Pattern* features always improve precision when combined with other features (at the expense of recall, unless they are combined with *n-grams*). The highest F_1 score (*i.e.*, 89.4%–85.9% precision and 93.5% recall) is achieved by DeMIBuD-ML using *pattern* features. We consider this version and configuration of DeMIBuD as the best for detecting missing EB.

DeMIBuD-ML detects missing S2R with recall ranging between 75.8% and 83.4%. These recall values are lower than that achieved by DeMIBuD-R. DeMIBuD-ML achieves lower precision than DeMIBuD-H, *i.e.*, between 60.8% and 69.2%. However, DeMIBuD-ML represents the best compromise, achieving the highest F_1 score (*i.e.*, 74.9%–69.2% precision and 83% recall) when using the *patterns + n-gram* features. Once again, among the different features used by DeMIBuD-ML, we observe that individual *n-grams* are the features that lead to the highest recall (*i.e.*, 83.4%) and always improve it when combined with other features. Conversely, *POS tags* are the features that lead to the lowest recall (*i.e.*, 75.8%) and always deteriorate it when combined with other features. DeMIBuD-ML based on *POS tags* achieves the lowest precision (*i.e.*, 60.8%), while *n-grams* lead to the highest (*i.e.*, 66.4%) and always improve it when combined with other features. Although individual pattern features lead to lower precision (*i.e.*, 63.5%), they always lead to precision improvement when combined with other features. The highest F_1 score (*i.e.*, 74.9%–69.2% precision and 83% recall) is achieved by DeMIBuD-ML using *patterns + n-gram* features. We consider this configuration of DeMIBuD as the best for detecting missing S2R.

Explaining the effect of individual features on the results is harder than with heuristics or regular expressions. However, we conjecture that the positive effect of *n-grams* is its ability to capture the vocabulary and (to some extent) the structure of EB/S2R and non-EB/S2R discourse. Our *patterns* also capture such characteristics; however, they further capture the discourse structure. POS tags focus on capturing the type of vocabulary and (to some extent) the structure, which has a negative impact on DeMIBuD-ML's accuracy. In any case, all features are insufficient to resolve content ambiguity, especially regarding S2R. As part of our future work, we plan to address this problem by capturing semantic properties of the text, via semantic frames [15] or rhetorical relations [21].

Table 5: Overall *cross-project* accuracy of DeMIBuD-ML.

Features	EB (Avg.)			S2R (Avg.)		
	Prec.	Recall	F_1	Prec.	Recall	F_1
pos	67.4%	94.3%	77.8%	60.1%	73.9%	63.8%
n-gram	77.9%	96.4%	86.0%	68.2%	86.3%	75.0%
pos + *n*-gram	76.5%	96.1%	85.1%	66.3%	86.5%	73.5%
patterns	87.3%	92.3%	89.5%	64.9%	89.1%	73.9%
patt. + pos	81.9%	92.2%	86.3%	63.7%	84.2%	71.5%
patt. + *n*-gram	86.9%	92.5%	89.5%	68.3%	87.5%	76.0%
all features	85.2%	92.6%	88.7%	69.2%	82.5%	74.4%

3.5.4 ***DeMIBuD-ML's Cross-Project Accuracy.*** Our machine learning-based DeMIBuD achieves the best F_1 score, but relies on supervised training. Obtaining training data from a project often poses challenges, so using training data from other projects is often desirable. We analyze DeMIBuD-ML's accuracy when bug reports from different projects are used to train its underlying learning model. We compare DeMIBuD-ML's accuracy when using *cross-project* (see Table 5) and *within-project* training (see Table 4).

In the case of EB, using *cross-project* training, we observe that DeMIBuD-ML's precision improves for all type of features (except for *pos*)—*i.e.*, 3% avg. improvement[3]. Conversely, DeMIBuD-ML's recall decreases 1.2% on average, except for *pos* and *pos + n-grams*. In the case of S2R, we observe that DeMIBuD-ML's precision improves for some features (*i.e.*, *n-gram*, *pos + n-gram*, *patterns*, and all features combined) and deteriorates for others (*i.e.*, *pos*, *patterns + pos*, and *patterns + n-gram*). We observe little precision improvement on average (*i.e.*, 0.4%). Instead, DeMIBuD-ML's recall improves substantially for most features (except for *pos*)—*i.e.*, 4.5% avg. improvement. The *patterns* features improve precision (*i.e.*, by 1.4%) and achieve the highest recall improvement among all features (*i.e.*, 8.8%). Overall, DeMIBuD-ML's accuracy is higher when using *cross-project* training than when using *within-project* training. One likely explanation is that the larger training data used in the *cross-project* training includes more patterns.

Remarkably, DeMIBuD-ML based on *patterns + n-gram* has the best accuracy[4] for both EB and S2R (see Table 5). Its high *cross-project* accuracy indicates that DeMIBuD-ML is extremely robust to the training strategy, and can be highly useful in a practical setting where labeled data from a new project is unavailable. This means that we can deploy DeMIBuD-ML in different projects (to the ones we used) without retraining and expect similar accuracy levels.

4 THREATS TO VALIDITY

The main threat to *construct validity* is the subjectivity introduced in *discourse patterns* extraction and in the construction of the labeled bug reports (Section 2.3.1). To minimize subjectivity, we ensured that each bug report was coded by two coders independently. We assessed coding reliability by measuring the inter-rater agreement

[3]The avg. improvement is computed by averaging the differences between the *cross-* and *within-project* precision/recall values, across the different types of features.
[4]While individual *patterns* and *patterns + n-gram* features lead to the same F_1 score, the latter ones are preferred because they lead to a slightly higher recall.

(Section 2.4.1). Regarding pattern extraction, our coding procedure was based on open coding practices [49] that aimed at minimizing subjectivity. The five coders extracted the patterns in a strict, iterative, and open manner [49], which led to continuous discussion of ambiguous cases, refinement of our pattern catalog and coded data, and assessment of our coding process. We also defined coding criteria and trained the coders on them via interactive tutorials.

To strengthen the *internal validity*, we mitigated the effect of different design and experimentation decisions (*e.g.*, text preprocessing) by tuning our three instances of DeMIBuD on data sets different from the ones used to measure DeMIBuD's accuracy.

To strengthen the *external validity*, we collected bug reports from nine software projects that cover different types of systems (*e.g.*, desktop, web, or mobile) and domains (*e.g.*, web-browsing or development). These projects are open source (except for Facebook), and use different bug trackers. The collected bug reports cover different types of bugs (*e.g.*, crashes or functional [67])—the distribution of bug types can be found in our replication package [23].

5 RELATED WORK

Our research relates to work on analysis of textual content, characterization and classification of issues, and issue quality assessment.

Analysis of Textual Content. Our work is based on automated discourse analysis. We followed the methodology proposed by Polanyi [57] for the analysis of the linguistic structure of discourse. We built on this analysis to identify discourse patterns based on grounded theory practices [49], *e.g.*, open coding. This technique has been extensively used in SE to, *e.g.*, identify types of knowledge in API documents [45], API privacy policy information [18, 64], or information relevant to development activity summaries [69].

Characterization of Issues. Issue (or bug) descriptions have been characterized from different angles and for different purposes. Previous work (*e.g.*, [16, 17, 26, 62, 63, 79]) focused on determining the structure of bug reports and its importance in bug triaging. Chilana *et al.* [24] investigated unwanted behavior types in bug reports. Tan *et al.* [67] identified defect types from bug reports. Breu *et al.* [19] determined stakeholders' information needs from bug reports. Ko *et al.* [38] analyzed bug report discussions to reveal software design decisions. Based on bug descriptions, Guo *et al.* [31] investigated which bugs get fixed. Ko *et al.* [37] and Rodeghero *et al.* [61] studied the role of different users in bug reporting.

Other work has focused on the textual characteristics of bug reports. Ko *et al.* [39] performed a linguistic analysis of bug report titles to understand how users describe software problems. Sureka *et al.* [66] analyzed the part-of-speech and distribution of words in issue titles to find vocabulary patterns useful in predicting the bug severity specified in bug reports. Chaparro *et al.* [22] and Moreno *et al.* [52] measured the vocabulary agreement between duplicate bug reports and between bug reports and source code, respectively. Different from existing work, our focus is on identifying the OB, EB, and S2R discourse used in bug descriptions.

Classification of Issues. Our research relates to work on issue classification [13, 31, 68, 72, 76], which relies on machine learning and textual features to classify issues as (for example) features requests, enhancements, or bug reports. Similar approaches have been proposed to classify e-mails [27], app reviews [48], forums [75], explanations of APIs in tutorials [56], and, outside SE, discourse elements in essays [20, 21]. The essential difference between our (SVM-based) approach and existing software content classifiers is the use of discourse patterns from bug descriptions. More related to our research is Davies *et al.*'s work [26], which proposed the explicit use of search terms (*e.g.*, *"observed behavior"*) to detect OB, EB, or S2R content in bug reports. Unfortunately, this approach produces numerous undetected cases (*i.e.*, false negatives).

Assessment and Improvement of Issue Quality. Our work also relates to research on issue quality assessment. Zimmerman *et al.* [78] proposed an approach to predict the quality level of bug reports. Dit *et al.* [28] and Linstead *et al.* [43] measured the semantic coherence in bug report discussions. Hooimeijer *et al.* [33] measured quality properties of bug reports (*e.g.*, readability) to predict when a bug report would be triaged. Zanetti *et al.* [74] identified valid bug reports, as opposed to duplicate, invalid, or incomplete reports, by relying on reporters' collaboration information. To enhance bug reports, Moran *et al.* focused on augmenting S2R in bug reports via screenshots and GUI-component images [51], and on automatically reporting potential crashes in mobile applications [50]. In another direction, some research has focused on summarizing bug reports [44, 46, 59] and detecting duplicate issues [22, 32, 42, 53]. Similar to these approaches, our final goal is to improve bug report quality. Our strategy, however, is to determine when essential information is absent from bug reports, and to alert users about it.

6 CONCLUSIONS AND FUTURE WORK

Our analysis of 2,912 bug reports from nine software systems revealed that while most of the reports (*i.e.*, 93.5%) describe OB, only 35.2% and 51.4% of them explicitly describe EB and S2R. These findings motivate our effort in developing an automated technique to detect the absence of EB and S2R in bug descriptions. In addition, from our discourse analysis of a subset of 1,091 bug report descriptions, we found that reporters recurrently use 154 discourse patterns to describe OB, EB, and S2R, and few of them (*i.e.*, 22 or 14.3%) appear in most of the bug reports that contain such information (*i.e.*, 82% on average). These results indicate that OB, EB, and S2R content can be automatically detected with high accuracy.

Based on the discourse patterns, we developed DeMIBuD, an automated approach that detects missing EB and S2R in bug descriptions. We implemented and evaluated three versions of DeMIBuD based on regular expressions, heuristics and NLP, and machine learning. Our ML-based approach (*i.e.,* DeMIBuD-ML) proved to be the most accurate in terms of F_1 score (*i.e.*, 89.4% for EB, and 74.9% for S2R), yet the other versions of DeMIBuD achieve comparable accuracy without the need for training. DeMIBuD-ML proved to be robust with respect to *within-* and *cross-project* training, which means that we can deploy it in different projects (to the ones we used) without retraining and achieve high accuracy detection. Our future work will focus on (i) studying acceptable recall/precision trade-offs from the DeMIBuD users' perspective, and (ii) addressing bug content ambiguity to improve DeMIBuD's accuracy.

Acknowledgments. The authors thank all the people that participated in the bug coding process. Chaparro and Marcus were supported in part by the NSF grant CCF-1526118. Bavota was supported in part by the SNF project JITRA, No. 172479.

REFERENCES

[1] 2016. An open letter to GitHub from the maintainers of open source projects. Available online: https://github.com/dear-github/dear-github. (2016).
[2] 2017. http://hibernate.org/. (2017).
[3] 2017. http://httpd.apache.org/. (2017).
[4] 2017. http://openmrs.org/. (2017).
[5] 2017. https://apps.wordpress.com/mobile/. (2017).
[6] 2017. http://search.cpan.org/~creamyg/Lingua-Stem-Snowball-0.952/lib/Lingua/Stem/Snowball.pm. (2017).
[7] 2017. https://www.docker.com/. (2017).
[8] 2017. https://www.facebook.com/. (2017).
[9] 2017. https://www.libreoffice.org/. (2017).
[10] 2017. https://www.mozilla.org/en-US/firefox/. (2017).
[11] 2017. http://www.eclipse.org/. (2017).
[12] Karin Aijmer and Anna-Brita Stenström. 2004. *Discourse Patterns in Spoken and Written Corpora*. Vol. 120. John Benjamins Publishing.
[13] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. 304–318.
[14] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. 2011. Extracting Structured Data from Natural Language Documents with Island Parsing. In *Proceedings of the International Conference on Automated Software Engineering (ASE'11)*. 476–479.
[15] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. 1998. The Berkeley Framenet Project. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and International Conference on Computational Linguistics (ACL'98)*. 86–90.
[16] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate Bug Reports Considered Harmful ... Really?. In *Proceedings of the International Conference on Software Maintenance (ICSM'08)*. 337–345.
[17] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting Structural Information from Bug Reports. In *Proceedings of the International Working Conference on Mining Software Repositories (WCRE'08)*. 27–30.
[18] Jaspreet Bhatia, Travis D. Breaux, and Florian Schaub. 2016. Mining Privacy Goals from Privacy Policies Using Hybridized Task Recomposition. *ACM Transactions on Software Engineering and Methodology* 25, 3 (2016), 1–24.
[19] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*. 301–310.
[20] Jill Burstein, Daniel Marcu, Slava Andreyev, and Martin Chodorow. 2001. Towards Automatic Classification of Discourse Elements in Essays. In *Proceedings of the Annual Meeting on Association for Computational Linguistics (ACL'01)*. 98–105.
[21] Jill Burstein, Daniel Marcu, and Kevin Knight. 2003. Finding the WRITE Stuff: Automatic Identification of Discourse Structure in Student Essays. *IEEE Intelligent Systems* 18, 1 (2003), 32–39.
[22] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. 2016. On the Vocabulary Agreement in Software Issue Descriptions. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*. 448–452.
[23] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Online replication package. (2017). https://seers.utdallas.edu/projects/missing-info-in-bugs
[24] Parmit K. Chilana, Andrew J. Ko, and Jacob O. Wobbrock. 2010. Understanding Expressions of Unwanted Behaviors in Open Bug Reporting. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*. 203–206.
[25] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.
[26] Steven Davies and Marc Roper. 2014. What's in a Bug Report?. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*. 26:1–26:10.
[27] Andrea Di Sorbo, Sebastiano Panichella, Corrado A. Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C. Gall. 2015. Development Emails Content Analyzer: Intention Mining in Developer Discussions (T). In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*. 12–23.
[28] Bogdan Dit, Denys Poshyvanyk, and Andrian Marcus. 2008. Measuring the Semantic Similarity of Comments in Bug Reports. In *Proceedings of International Workshop on Semantic Technologies in System Maintenance (STSM'08)*. 265–280.
[29] Geoff Dougherty. 2012. *Pattern Recognition and Classification: An Introduction*. Springer Science & Business Media.
[30] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for Me! Characterizing Non-reproducible Bug Reports. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'14)*. 62–71.
[31] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. 495–504.
[32] Abram Hindle, Anahita Alipour, and Eleni Stroulia. 2016. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering* 21, 2 (2016), 368–410.
[33] Pieter Hooimeijer and Westley Weimer. 2007. Modeling Bug Report Quality. In *Proceedings of the International Conference on Automated Software Engineering (ASE'07)*. 34–43.
[34] Thorsten Joachims. 1998. *Making Large-Scale SVM Learning Practical*. LS8-Report 24. Universität Dortmund, LS VIII-Report.
[35] Thorsten Joachims. 1998. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In *Proceedings of the European Conference on Machine Learning (ECML'98)*. 137–142.
[36] Thorsten Joachims. 2006. Training Linear SVMs in Linear Time. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD'06)*. 217–226.
[37] Andrew J. Ko and Parmit K. Chilana. 2010. How Power Users Help and Hinder Open Bug Reporting. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'10)*. 1665–1674.
[38] Andrew J. Ko and Parmit K. Chilana. 2011. Design, Discussion, and Dissent in Open Bug Reports. In *Proceedings of the iConference (iConference'11)*. 106–113.
[39] Andrew J. Ko, Brad A Myers, and Duen Horng Chau. 2006. A Linguistic Analysis of How People Describe Software Problems. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 127–134.
[40] Klaus Krippendorff. 2004. *Content Analysis: An Introduction to Its Methodology* (2nd ed.). Sage.
[41] Eero I. Laukkanen and Mika V. Mäntylä. 2011. Survey Reproduction of Defect Reporting in Industrial Software Development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'11)*. 197–206.
[42] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. 2016. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software* 121 (2016), 223–233.
[43] Erik Linstead and Pierre Baldi. 2009. Mining the Coherence of Gnome Bug Reports with Statistical Topic Models. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR'09)*. 99–102.
[44] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. 2014. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Software Engineering* 20, 2 (2014), 516–548.
[45] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.
[46] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. AUSUM: Approach for Unsupervised Bug Report Summarization. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE'12)*. 11:1–11:11.
[47] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'14)*. 55–60.
[48] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. 2016. A Survey of App Store Analysis for Software Engineering. *IEEE Transactions on Software Engineering* (to appear), 99 (2016).
[49] Matthew B. Miles, A. Michael Huberman, and Johnny Saldaña. 2013. *Qualitative Data Analysis: A Methods Sourcebook* (3rd ed.). SAGE Publications, Inc.
[50] Kevin Moran, Mario Linares-Váquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'16)*. 33–44.
[51] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing Bug Reports for Android Applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*. 673–686.
[52] Laura Moreno, Wathsala Bandara, Sonia Haiduc, and Andrian Marcus. 2013. On the Relationship between the Vocabulary of Bug Reports and Source Code. In *Proceedings of the International Conference on Software Maintenance (ICSM'13)*. 452–455.
[53] Anh Tuan Nguyen, Tung Thanh Nguyen, Tuan N Nguyen, Daniel Lo, and Chengnian Sun. 2012. Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling. In *Proceedings of the International Conference on Automated Software Engineering (ASE'12)*. 70–79.
[54] Paul Ogilvie and James P. Callan. 2001. Experiments Using the Lemur Toolkit. In *TREC*, Vol. 10. 103–108.
[55] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. 2016. ARdoc: App Reviews Development Oriented Classifier. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'16)*. 1023–1027.

[56] Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. 2015. Discovering Information Explaining API Types Using Text Classification. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*. 869–879.

[57] Livia Polanyi. 2003. *The Handbook of Discourse Analysis*. Vol. 18. Wiley-Blackwell, Chapter The Linguistic Structure of Discourse, 265.

[58] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. 2015. StORMeD: Stack Overflow Ready Made Data. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'15)*. 474–477.

[59] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic Summarization of Bug Reports. *IEEE Transactions on Software Engineering* 40, 4 (2014), 366–380.

[60] Peter C. Rigby and Martin P. Robillard. 2013. Discovering Essential Code Elements in Informal Documentation. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. 832–841.

[61] Paige Rodeghero, Da Huo, Tao Ding, Collin McMillan, and Malcom Gethers. 2016. An empirical study on how expert knowledge affects bug reports. *Journal of Software: Evolution and Process* 28, 7 (2016), 542–564.

[62] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. 485–494.

[63] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. 2016. What Makes a Satisficing Bug Report?. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'16)*. 164–174.

[64] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a Framework for Detecting Privacy Policy Violations in Android Application Code. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*. 25–36.

[65] Mozilla Support. 2017. Contributors Guide to Writing a Good Bug. (2017). https://goo.gl/ykkrUJ

[66] A. Sureka and P. Jalote. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *Proceedings of the Asia Pacific Software Engineering Conference (ASPEC'10)*. 366–374.

[67] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical Software Engineering* 19, 6 (2014), 1665–1705.

[68] Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Automatic Defect Categorization. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*. 205–214.

[69] Christoph Treude, Fernando Figueira Filho, and Uirá Kulesza. 2015. Summarizing and Measuring Development Activity. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*. 625–636.

[70] Anthony J. Viera, Joanne M. Garrett, et al. 2005. Understanding Interobserver Agreement: The Kappa Statistic. *Family medicine* 37, 5 (2005), 360–363.

[71] Dhaval Vyas, Thomas Fritz, and David Shepherd. 2014. Bug Reproduction: A Collaborative Practice within Software Maintenance Activities. In *Proceedings of the International Conference on the Design of Cooperative Systems (COOP'14)*. 189–207.

[72] Bowen Xu, David Lo, Xin Xia, Ashish Sureka, and Shanping Li. 2015. EFSPredictor: Predicting Configuration Bugs with Ensemble Feature Selection. In *Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC'15)*. 206–213.

[73] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2016. Learning to Extract API Mentions from Informal Natural Language Discussions. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*. 389–399.

[74] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. 1032–1041.

[75] Bo Zhou, Xin Xia, David Lo, Cong Tian, and Xinyu Wang. 2014. Towards More Accurate Content Categorization of API Discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC'14)*. 95–105.

[76] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28, 3 (2016), 150–176.

[77] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. 1074–1083.

[78] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.

[79] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. 2009. Improving Bug Tracking Systems. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*. 247–250.