The Journal of Systems & Software 202 (2023) 111709

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A first look at bug report templates on GitHub ☆

Hongyan Li, Meng Yan *, Weifeng Sun, Xiao Liu, Yunsong Wu

School of Big Data and Software Engineering, Chongqing University, Chongqing, China

ARTICLE INFO

Article history:
Received 31 July 2022
Received in revised form 22 February 2023
Accepted 14 April 2023
Available online 23 April 2023

Keywords:
Bug reports
Template
Empirical study

ABSTRACT

Bug reports which are written by different people have a variety of styles, and such various styles lead to difficulty in understanding bug reports. To enhance the comprehensibility of bug reports, GitHub has proposed a template mechanism to guide how to report the bugs. However, there is no study on the use of bug report templates on GitHub. In this paper, we conduct an empirical study on the bug report templates on GitHub, including the popularity, benefits, and content of the templates. Our empirical study finds that: (1) For popularity, more and more open source projects and bug reports are applying templates over time. (2) For benefits, bug reports written using templates will be resolved quicker and have a higher comment coverage. (3) For content, the most common items for templates are *expected behavior*, *describe the bug* and *to reproduce* etc. Additionally, we summarize a taxonomy of items for bug report templates. Finally, we propose an automatic templating approach for templating an un-templated bug report. Our approach achieves an accuracy of 0.718 and an F1-score of 0.717 on average, which shows that our approach can effectively templatize an un-templated bug report.

1. Introduction

Writing programs without any bugs is difficult for developers and practitioners. Users and developers report problems when it does not behave as expected (Zimmermann et al., 2009). Specifically, problems are often reported in the form of bug reports. A bug report is a document that describes the bug and contains the information needed to fix the bug (Nguyen et al., 2011). The rapid development of software (especially open source software) has led to an increasing number of bug reports (Anvik et al., 2005). For example, GitHub, the world's leading software development platform, has more than 20+ million issues in 2019 just in closed status (GitHub, 2019). And creating an issue is the most common thing that people do in their first hour on GitHub (GitHub, 2021). Due to the large scale of bug reports, handling them can be challenging for developers.

Bug reports are written in natural language by different users or developers (Zhang et al., 2019). Due to the different styles and expertise of different reporters, bug reports suffer from various styles and are difficult to understand. For example, we find that there are more than 61 and 39 kinds of phrase expressions for describing *the environment of the bug* and *steps to reproduce the bug*, respectively.

```
---
Name: Bug report
About: Something is crashing or not working as intend
Labels: bug
---
**Library Version:**
2.x.x

**Affected Device(s):**
Google Pixel 3 XL with Android 9.0

**Describe the Bug:**
A clear description of what is the bug is.

**To Reproduce:**
Provide all necessary steps to reproduce the bug.

**Expected Behavior:**
A clear description of what you expected to happen.
```

Fig. 1. An example of a bug report template on GitHub.

To enhance the comprehensibility of bug reports, GitHub has proposed a template mechanism, where each project can set a specific template to guide how to describe the bugs. Fig. 1 shows an example of a bug report template on GitHub. The bug report template consists of a set of items and explanations (indicated by underlined text) corresponding to the items, where the explanations are statements or examples of what the items

☆ Editor: Martin Pinzger.

* Corresponding author.
E-mail addresses: hongyan.li@cqu.edu.cn (H. Li), mengy@cqu.edu.cn (M. Yan), weifeng.sun@cqu.edu.cn (W. Sun), cdjx@cqu.edu.cn (X. Liu).

https://doi.org/10.1016/j.jss.2023.111709
0164-1212/

mean. However, the use of bug report templates such as the popularity, benefit, and content has never been investigated. In this paper, we conduct the first empirical study on the bug report templates (simplified as templates) on GitHub. We start with understanding the popularity of templates, then explore their benefits and analyze their content. Additionally, we propose an automatic templating approach for templating an un-templated bug report (i.e., bug reports written without using templates). The study discusses the following research questions in particular.

RQ1: How common do projects use templates on GitHub? We carefully select 1000 of the most popular GitHub projects. We count projects with and without bug report templates and analyze the use of templates over time. We find that more and more projects and bug reports are using templates over time.

RQ2: What are the benefits of using templates? We answer this question in two ways: resolution time cost and comment coverage of a bug report. We calculate the resolution times and comment coverage for bug reports with and without templates in each project to explore this question. We find that bug reports with templates have shorter resolution times and higher comment coverage, proving that using templates to report a bug is beneficial.

RQ3: What are the items of template content? To answer this question, we first investigate the most common items, and then explore the main item categories by merging semantic-similar items (e.g., *describe the bug* and *bug description*) according to their explanations. A category consists of a set of similar items. As a result, the most commonly used items are *expected behavior*, *describe the bug*, *to reproduce*, *additional context*, *environment*, *screenshots*, *steps to reproduce*, *desktop*, *actual behavior*, *smartphone*, *current behavior*, and *issue description*. And we summarize a taxonomy of items with 11 categories, e.g., *Description*, *Reproduction*, *Expected Behavior*.

RQ4: How to templatize a bug report? We propose an automatic templating approach for templating an un-templated bug report. The results show that we achieve an average accuracy of 0.718 and an average F1-score of 0.717.

This paper makes the following contributions:

- We provide insight into the popularity and benefits of bug report templates, which reflects the importance of bug report templates.
- We make a comprehensive understanding of template content by exploring the most common items and the main item categories.
- We propose an automatic templating approach. The results show that our approach can effectively templatize an un-templated bug report.
- We open source our replication package (Our Replication Package, 2023) for further studies, including the dataset and the source code of our study.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 describes the dataset. We present the methods and answers to each of the four research questions from Sections 4 to 7. Section 8 summarizes the main threats of our study. Section 9 discusses the evaluation of un-templated bug reports and the implications of our study. The conclusion is concluded in Section 10.

2. Related work

In this section, we introduce the related work about content optimization and comprehension of bug reports.

2.1. Content optimization of bug reports

Providing as much detail as possible in the bug report would be laborious and impractical, so it is important to optimize the content of the bug report.

For bug report fields, Bettenburg et al. (2007) investigate the quality of bug reports for the first time by surveying Eclipse developers. The results show that *steps to reproduce* and *stack traces* are the most important information in bug reports, but *steps to reproduce* would have the worst impact if it contains error information. In addition, they provide a content-based tool for assessing the quality of bug reports. Xie et al. (2013) find that triager improves the quality of bug reports in three ways: filtering reports, identifying related products, and adding the missing information. Moreover, Breu et al. (2010) suggest users include the fields *screenshot*, *stack trace*, and *steps to reproduce* in their bug reports by investigating bug reports from Mozilla and Eclipse. To predict the wrong components of the bug report, Lamkanfi and Demeyer (2013) propose a data mining technique to solve it. Considering the complexity of Bugzilla reports, Herraiz et al. (2008) decrease the number of severity options for bug reports from seven to three.

For the whole bug report, Wu et al. (2011) find that bug reports are often incomplete and propose BUGMINER to detect the completion of bug reports. Xia et al. (2014) find that fixing a modified bug report takes more time through an empirical study of open source projects.

2.2. Comprehension of bug reports

Bug reports are primarily based on natural language, so their comprehensibility is critical.

There are some previous works on the summarization of bug reports. Rastkar et al. (2010) first propose to generate summaries for bug reports because they believe that an appropriate summary would aid developers in rapidly perusing and locating the right bug report when analyzing existing bug reports. In addition, Rastkar et al. (2014) build a supervised-based summarizer to generate summaries automatically. Jiang et al. (2019) propose a supervised approach to summarize bug reports using crowd-sourced elicited attributes. Due to the lack of summaries as labels, many works also build unsupervised approaches to summarize the bug reports based on classic unsupervised summarization approaches (Mani et al., 2012), heuristic rules (Lotufo et al., 2015), and auto encoder (Li et al., 2018). Mani et al. (2012) evaluate four unsupervised techniques for bug report summarization and improve the efficacy of the unsupervised by applying a noise identifier and filtering sentences. Lotufo et al. (2015) propose a graph-based unsupervised approach inspired by human reading behaviors. Li et al. (2018) propose a stepped auto-encoder unsupervised network, which successfully integrates the bug report characteristics into a deep neural network.

In addition, Ko et al. (2006) investigate the way people describe software bugs by analyzing bug report headings and proposed more structured report forms based on this. Chen et al. (2020) propose an approach named ITAPE, which automatically generates headings for bug reports.

3. Dataset

We describe how we build our dataset, including how we select and filter projects and how we identify bug reports with templates.

3.1. Project selection

This paper is based on data from GitHub, an open source software project hosting platform where users can manage their project code and browse other users' open project code. When bugs are found in open source projects, users can submit bug reports in the issues module to inform developers and other users. In addition, software developers can also discuss and communicate in the issues module to submit their views in the comment module. To ensure a broad and random dataset, we select the top 1000 most popular projects (with the highest number of stars) on GitHub, which helps to (1) collect bug reports posted by different reporters with different writing styles (2) and cover a wide range of project types. Actually, not all projects provide templates, 99 projects have English bug report templates and 5 projects have Chinese templates. For generalizability, we only consider templates written in English. Thus, our final dataset consists of such 99 projects with English templates.

Finally, we obtain all the templates and issues (issue number, issue label, issue status, created time, closed time, body, and number of comments) of the 99 projects with templates.

3.2. Bug reports and templates identification

3.2.1. Identification of bug reports from issues

GitHub gives project members the right to organize and prioritize their work. They can label issues with categories or any other helpful information, e.g., "bug", "dependence", "environment", "question", and "ui". GitHub encourages projects to tag the "bug" for issues that report a bug (GitHub, 2022). Therefore, we keep the issues labeled with "bug", "Bug", and "BUG" as bug reports.

3.2.2. Identification of bug reports written using templates

Generally, many projects give templates according to their project characteristics and require the bug report writers to submit bugs according to the templates. The bug report template usually consists of a set of items and explanations corresponding to the items. If a bug report template is used to report a bug, the bug report writer populates the template based on the items in the template. However, in practice, a large number of bug reports do not follow or partially follow the template to write bug reports. In other words, (1) different projects have different templates, and different templates have different items, (2) bug reports for projects with templates are not always written using the templates, and (3) bug reports that use templates do not necessarily use all the items in the templates. Therefore, for a project with a template, if a bug report uses an item from the template, we assume that the bug report is written using the template.

Finally, we obtain 63,105 bug reports from 99 projects with templates, including 18,058 bug reports written using templates and 45,047 bug reports written without using templates.

4. RQ1: How common do projects use templates on GitHub?

We aim to investigate the popularity of bug report templates on GitHub. All the analyses are conducted quantitatively.

4.1. Methodology

We answer this RQ in two ways: (1) the trend of application overall. (2) the trend of application within projects. We first explore the trend of using templates for projects and bug reports in general, and then explore the trend of using templates in each project. The analysis is based on the projects and bug reports described in Section 3.

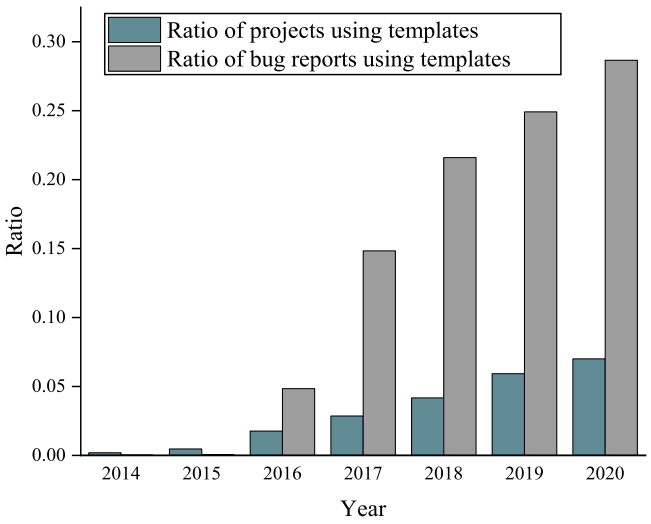


Fig. 2. Trend of the using templates for all projects and templates.

Trend of application overall. We conduct a statistical analysis at two levels to study the trends of using templates: projects and bug reports. For bug reports, when it uses an item from the project template, we consider it uses the template, as described in Section 3.2.2. For projects, we consider whether the project uses the provided template rather than just providing it. Some projects provide templates, but the templates are not used by any of the bug reports in the project (e.g., "feathericons/feather"). Therefore, we consider a project to start using the template when there is a bug report in the project written using the template. To show trends of using templates, we calculate the following two indicators from 2014 to 2020: **#Ratio of project using templates:** As of this year, the ratio of the number of projects using templates to the number of all projects created. **#Ratio of bug reports using templates:** As of this year, the ratio of the number of bug reports using the templates to the number of all bug reports created.

Trend of application within projects. To explore the trend of using templates within projects, we conduct further research on projects that provided templates. We count the ratio of templated bug reports for each project and show the ratio distribution for all projects from 2014 to 2020. In this case, the ratio of bug reports using templates in each project in each year is calculated as follows: the ratio of the number of bug reports using templates to the number of all bug reports created in the project in that year.

4.2. Results

Trend of application overall. Fig. 2 shows the ratio of projects and bug reports that use templates from 2014 to 2020. We can find that more and more projects are using templates over time. Similarly, more and more bug reports are written using templates. Especially in the last few years, it has shown a significant growth trend. In 2020, nearly 30% of the bug reports are written using templates. It suggests that bug report templates are becoming an increasingly popular way to report bugs.

Trend of application within project. Fig. 3 shows box plots from 2014 to 2020 on the ratio of bug reports with templates for each project. And each box plot shows the distribution of the ratio of bug reports with templates per project. The line in the graph shows the trend in the mean ratios for each year. We can find that as the years grow, the ratio of the per-project using the templates to write bug reports gets higher. In 2020, nearly

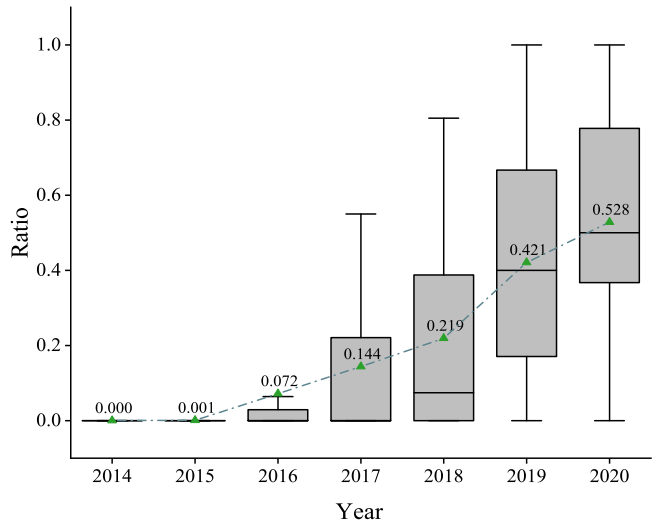


Fig. 3. Trend of the ratio of templated bug reports for each project with template.

52.8 percent of the bug reports per project are reported using templates. It suggests that there is a growing preference to use templates to report bug reports for projects with templates.

> *Summary:* On GitHub, more and more projects are using templates to report bugs, and more and more bug reports are written using templates. Similarly, for each project that has a template, more and more bug reports are written using templates.

5. RQ2: What are the benefits of using templates?

We intend to explore the benefits of bug reports written using templates, which is also the research motivation of this paper.

5.1. Methodology

The main purpose of a bug report is to help developers understand the bug and resolve it. Therefore, we answer this question in two ways: "Comment" and "Resolution". The first aspect focuses on the effect of improving attention and increasing discussion, while the second aspect focuses on the efficiency of bug resolution. To this end, we explore the benefits by comparing the metrics of these two aspects calculated separately for bug reports with and without templates in the project.

As the years increase, the GitHub community grows, and new projects and bug reports appear. For a more comprehensive statistical analysis, we first count the overall situation and then by year according to the creation time of bug reports in the project. To reduce errors in the metrics, we keep projects that have at least 5 bug reports written using templates. Similarly, when counting by year, to reduce statistical errors caused by the small number of bug reports written using templates created during the year in the project, we also include bug reports created before that year. And only projects with at least 5 bug reports written using templates as of that year are added to the calculation for that year. As of 2015, only 2 projects are using the template and the number of bug reports written using the template is less than 5 in each of them, so our statistics start in 2016. For each project, we calculate the following metrics separately for bug reports with templates and those without templates.

5.1.1. Comments

Some researchers have found that the comments are good for resolving the bug (Hooimeijer and Weimer, 2007; Panjer, 2007; Ramirez-Mora et al., 2021). Hooimeijer and Weimer (2007) observed that bug reports with many comments are resolved more quickly in Firefox, suggesting that bugs that receive more user attention are fixed faster. Panjer (2007) observed for Eclipse that comment count affects the lifetime the most. And Ramirez-Mora et al. (2021) found that the average number of comments for bugs that were fixed was greater than the average number of comments for bugs that were not fixed. Therefore, we calculate two metrics related to the comments for the bug reports with and without templates in the project.

#Comment Number: The average number of comments for bug reports with comments in a project.

#Comment Coverage: The ratio of the number of bug reports covered by comments to the total number of bug reports in a project. Notably, for a bug report, if the number of comments is greater than zero, the bug report is covered.

5.1.2. Resolution

Obviously, bug reports written using templates are more readable, and the more readable bug reports are fixed faster (Bettenburg et al., 2008). So the resolution time can be a good measure of the quality of a bug report, which also widely adopted in previous work (Datta et al., 2021; Datta and Lade, 2015; Datta et al., 2014; Abreu et al., 2021; Zhang et al., 2013). Therefore, we calculate the resolution times for the bug reports with and without templates in the project.

#Resolution Time: The average resolution time for resolved bug reports in a project. For a bug report, the resolution time is the time cost (measured in days) from "created time" to "closed time" of the bug report.

To explore whether closed bug reports are actually resolved (especially those that have been open for a relatively long time to close), we randomly select 10 bug reports that have been closed for more than 400 days for manual analysis. By manually checking and analyzing their comments and commits, we find that 90% of the bug reports are actually resolved. This indicates that most of the bug reports are closed only after they are resolved, which is why a few of them take a relatively long time to close. For example, the bug report "#6134" for the project "sequelize/sequelize" is created on June 19, 2016, and fixed and closed after 601 days on February 11, 2018, via "Pull Request #9033". During the time it is open, 16 different developers comment one after another that they encounter the same bug, even two days before the bug is fixed. And the bug report "#8044" for the project "elastic/elasticsearch" was created on October 10, 2014, and closed on August 24, 2016, after 684 days by jpountz, who comments that the bug is fixed in master. Moreover, the ratio of bug reports that take more than 200 days or 400 days to resolve is small. The ratio of bug reports with templates that take 200 to 400 days to resolve is 0.080, and only 0.038 for those that take more than 400 days. The ratio of bug reports without templates that take 200 to 400 days to resolve is 0.051, and only 0.062 for those that take more than 400 days.

5.2. Results

5.2.1. Comments

Results of overall statistics. Table 1 shows the overall statistical difference of comments for bug reports with and without templates in the projects. We can find that the average comment coverage for bug reports with templates is higher than for those without templates. And the average number of comments for commented bug reports is also higher when a template is

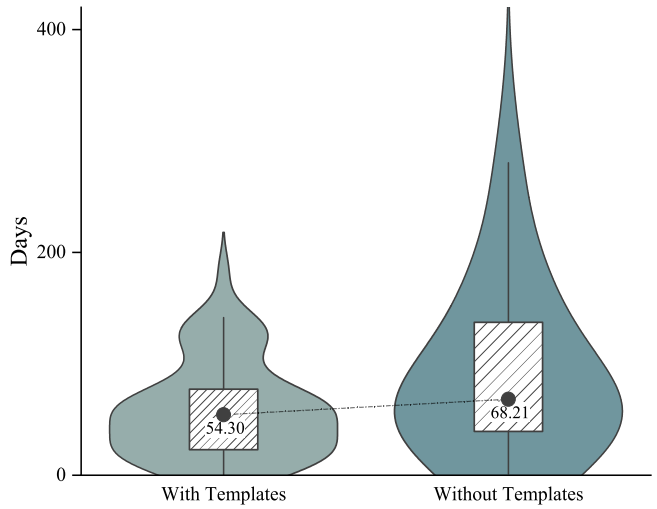


Fig. 4. The overall comparison of resolution times between bug reports with and without templates.

Table 1
The overall statistical difference of comments for bug reports with and without templates in our dataset.

	Mean	
	With templates	Without templates
#Comment coverage (%)	0.923	0.895
#Comment number	6.994	6.838

Table 2
The annual statistical difference of comments for bug reports with and without templates in our dataset (Mean).

Year	#Comment coverage (%)		#Comment number	
	With templates	Without templates	With templates	Without templates
2016	**0.944**	0.877	**8.448**	6.252
2017	**0.936**	0.879	**8.247**	6.788
2018	**0.944**	0.889	**8.205**	7.060
2019	**0.932**	0.898	**7.745**	7.015
2020	**0.923**	0.895	**6.994**	6.838

used. This shows that a clearly structured bug report written using a template can, to some extent, gain the attention of more developers and get more discussion.

Results of statistics by year. Table 2 shows the statistical difference of comments for bug reports with and without templates in the projects from 2016 to 2020. It can be seen that the average comment coverage of bug reports with templates is higher than for those without templates each year. For bug reports with comments, the average number of comments for bug reports with templates is also higher each year than for bug reports without templates.

5.2.2. Resolution

Results of overall statistics. Fig. 4 shows the distribution differences between bug reports with and without templates in their resolution times. The left and right violin plots show the distribution of the average resolution time for bug reports with and without templates per project, respectively. We can find that the box plots of the bug reports with templates have shorter upper whiskers, suggesting that the distribution of the average resolution time for bug reports with templates is relatively concentrated. Moreover, we can find that the box plots for bug reports with templates have lower box cap and floor, and the point in the graph representing the median of the resolution times is lower with the template than without. All this means that bug reports with templates are more likely to have a shorter resolution time. Table 3 shows the statistical difference of resolution for bug reports with and without templates in the projects. It can be found that the average resolution time of bug reports with templates is reduced by 47.126 days on average and 13.913 days on median compared to those without templates, which shows that bug reports with templates are more efficient and take less time to resolve bugs.

Table 3
The overall statistical difference of average resolution time for bug reports with and without templates in our dataset.

	Median (days)	Mean (days)
With templates (x)	54.295	58.559
Without templates (y)	68.208	105.685
Differences ($y - x$)	13.913	47.126

Table 4
The annual statistical difference of average resolution time for bug reports with and without templates in our dataset.

Year	Median (days)		Mean (days)	
	With templates	Without templates	With templates	Without templates
2016	107.015	75.797	167.947	132.698
2017	108.741	68.124	135.545	111.054
2018	100.677	78.543	135.960	112.977
2019	**71.488**	82.002	**80.702**	123.517
2020	**54.295**	68.208	**58.559**	105.685

Results of statistics by year. Fig. 5 shows the distribution differences between bug reports with and without templates in their resolution times from 2016 to 2020. The left and right sides of each violin plot show the distribution of resolution times for bug reports with and without templates per project. The solid line in the violin plot represents the median and the dashed line represents the quartiles. As can be seen, the distribution of resolution times for bug reports with templates is increasingly concentrated in a shorter range of resolution times as the year increases, but there is no significant change for bug reports without templates. And, starting in 2019, there is a significant difference in the distribution of resolution times for bug reports with and without templates: The distribution of bug reports with templates is more concentrated in a lower range of resolution times compared to those without templates. Moreover, starting in 2019, the median and quartiles of resolution times for bug reports with templates are shorter than for bug reports without templates. Table 4 shows the statistical difference in resolution for bug reports with and without templates in the projects from 2016 to 2020. We can find that starting from 2019, the median and mean of resolution times for bug reports with templates starts to be shorter than for bug reports without templates. In particular, by 2019, the average resolution time for bug reports with templates is 10.514 days shorter than those without templates on the median and 42.815 days shorter on the mean. By 2020, the average resolution time for bug reports with templates is 13.913 days shorter than those without templates on the median and 47.126 days shorter on the mean. It can be seen that as the years go by, the average resolution time for bug reports with templates is not only lower than those without templates, but the difference also becomes larger. This may be due to the fact that the ratio of bug reports written using templates is increasing year by year, as concluded in RQ1. By 2020, the number of new bug reports with templates in projects is comparable to the number of bug reports without templates in projects, and comparisons are relatively fair.

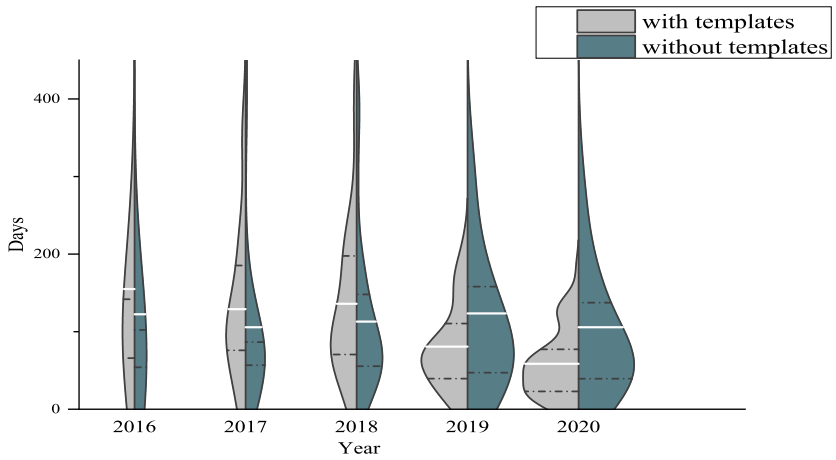


Fig. 5. The annual comparison of resolution times between bug reports with and without templates.

Summary: Bug reports written using templates not only gain higher comment coverage and more comments, but also take less time to be resolved.

6. RQ3: What are the items of template content?

We aim to explore what items are most commonly used in templates, and what are the main categories of items.

6.1. Methodology

After manually checking the templates in our dataset, we find that the bug report template for each project is written according to the characteristics of the project. The template items differ between projects, items with different representations may be the same semantic. For example, the item *what happen instead* has the same semantic meaning as the item *actual result*.

Therefore, we do the following steps to investigate the items: (1) Item quantity analysis. We first count the items of all bug report templates to explore which items are commonly used for templates. (2) Item category analysis. Based on the results of item quantity analysis, we use a two-iteration card sorting approach to aggregate all the items. Such an aggregation step aims to explore the main categories of items by merging semantic-similar items. Each category is representative of a class of semantic-similar items.

6.1.1. Item quantity analysis

We first extract all the items from bug report templates manually, then format and count the items.

Before we count all items, we do the following two steps:

Item Extraction: Different projects may use different formats for a same item in the template. For example, both "##environment*" and "*Environment**" represent the meaning of the environment, but the format is not the same. Extracting all the items automatically and accurately is difficult. Therefore, we manually check the template content to extract all the template items.

Item Formatting: We format the items before performing statistical analysis. In detail, we remove non-English characters from the items, such as "#", "*", and format all items in lowercase. For example, "##Environment" is formatted as "environment".

6.1.2. Item category analysis

In order to explore the taxonomy of the template items for bug reports, we need to classify items of all templates according to the corresponding explanations and finally generate a taxonomy of the items.

We use a two-iteration card sorting (Spencer and Donna, 2009) approach to classifying items. We create one card for each item, where the card contains the interpretation of the item and the context of the item.

Iteration 1. We define the initial categories by the semantic merging of common items based on item explanations, where the common items are the results of *Item Quantity Analysis*. Based on the principle of brevity and common usage, the initial category names are defined as follows: *Expected Behavior*, *Description*, *Reproduction*, *Additional Context*, *Environment*, *Screenshots*, and *Actual Behavior*. We ask two volunteers manually and independently classify all the items based on the initial item categories. After independent classifying, the two volunteers work together to discuss disagreements and items that could not be classified.

At the end of this iteration, 11.36% of the items still cannot be classified after the discussion between the two volunteers (e.g., *self-diagnosis*). This indicates that the initial categories we defined are deficient. So we need to adjust the item categories in *Iteration 2* based on unclassified items.

Iteration 2. The two volunteers then independently classify the remaining items by merging semantic-similar items together into a new category based on item interpretation. In order to show the category of template items more comprehensively, for the item categories with counts greater than 2, we keep them and name them based on concise and common usage. For the remaining unclassified items, each of which is an item category, we classify these items into the *Other* item category for visualization purposes. In addition, we classify those items that have the same expression but different meaning in the templates of different projects into *Other* item category as well.

After the classification, we discuss all item categories and suggest a common bug report template.

6.2. Results

6.2.1. Results for item quantity statistic

Table 5 shows the most common items for bug report templates, where "Count" represents the number of the item in all templates, "Ratio" represents the ratio of the number of the item to the number of all items. We do not list the items with a count less than 5, because their percentage is no more than 0.1%.

From Table 5, we can find that the commonly used items are as follows.

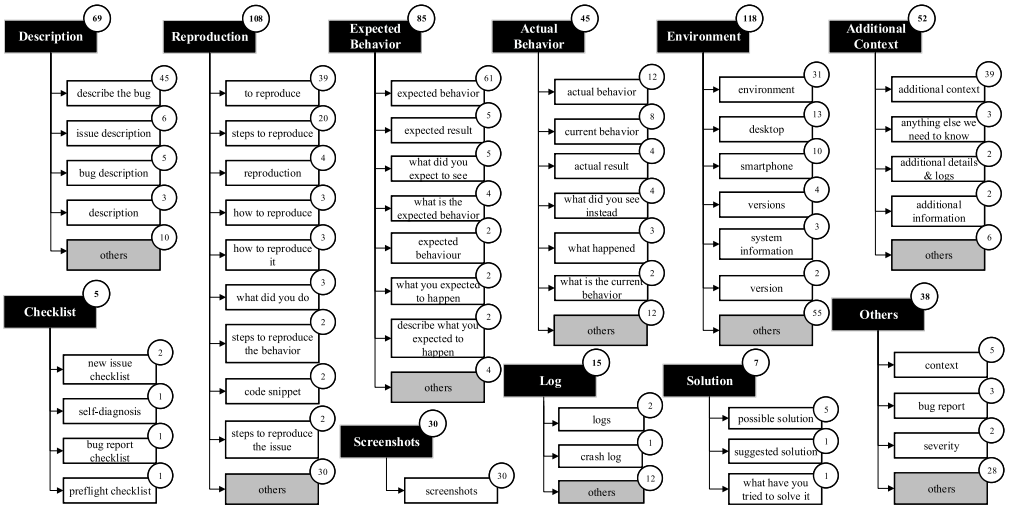


Fig. 6. Taxonomy of items for bug report templates.

Table 5
Statistics of the most common items.

Rank	ItemName	Count	Ratio
1	Expected behavior	61	10.66%
2	Describe the bug	45	7.87%
3	To reproduce	39	6.82%
4	Additional context	39	6.82%
5	Environment	31	5.42%
6	Screenshots	30	5.24%
7	Steps to reproduce	20	3.50%
8	Desktop	13	2.27%
9	Actual behavior	12	2.10%
10	Smartphone	10	1.75%
11	Current behavior	8	1.40%
12	Issue description	6	1.05%

- The item *expected behavior* guides to describe what would happen if the bug is not present.
- The items *describe the bug*, *issue description* guide to provide a brief description of the bug.
- The items *to reproduce*, *steps to reproduce* guide to describe how the bug can be reproduced.
- The item *additional context* guides to describe any other information related to the bug.
- The items *environment*, *desktop*, *smartphone* guide to describe the environment at the time of the bug, such as operating system or version information.
- The item *screenshots* guides to give screenshots related to the bug.
- The items *actual behavior*, *current behavior* guide to describe what actually happens when there is a bug.

6.2.2. Results for the taxonomy

Fig. 6 shows the taxonomy of items by our card sorting for bug report templates, which are composed of eleven root categories: *Description*, *Reproduction*, *Expected Behavior*, *Actual Behavior*, *Environment*, *Additional Context*, *Checklist*, *Screenshots*, *Log*, *Solution*, and *Others*. The semantic-similar items in each category are represented as leaves. In addition, the value in the circle box in the upper right corner shows the item count. As for the value of the box "others" in the category, it indicates the number of semantic-similar items with a count of 1.

For each category, we present the specific count (immediately following the item category name), and discuss it in detail. At the end of this section, we suggest a common bug report template.

***Description (69)*:** This category mainly guides to provide a brief description of the bug. There are 69 semantic-similar items in this category, accounting for 12.06% of all items. There are 14 expressions in this category, and more than half of them are *describe the bug (45)*. But for brevity, we define the category as *Description*. For a bug report, a brief bug description can help others quickly understand what the bug report is about.

***Reproduction (108)*:** This category mainly guides to describe the steps to reproduce the bug. There are 108 semantic-similar items with 39 expressions in this category, and it accounts for 18.88% of all items, making it the second largest category. For a bug report, describing how the bug happened so that others can deeply understand the bug and reproduce it.

***Expected Behavior (85)*:** This category mainly guides to describe what would happen without the bug. There are 85 semantic-similar items with 11 expressions in this category, and it accounts for 14.86% of all items, making it the third largest category. For a bug report, describing what would happen without the bug can help others explore the reasons for the bug.

***Actual Behavior (45)*:** This category mainly guides to describe what would happen with the bug. There are 45 semantic-similar items with 18 expressions in this category, and it accounts for 7.87% of all items. We can see that the count of *Actual Behavior* is less than *Expected Behavior*, because there are some templates where the *Description* may contain *Actual Behavior*. However, for a bug report, *Actual Behavior* and *Expected Behavior* are meaningful categories because it helps others to understand the gap between *Actual Behavior* and *Expected Behavior* in detail and explore the reasons why the bug happens from the gap in results.

***Environment (118)*:** This category mainly guides to describe where the bug happened. That is, it describes the environment in which the bug happened, e.g., *version*, *desktop*. There are 118 semantic-similar items with 61 expressions in this category, and 55 of these expressions have an item count of 1, which illustrates the diversity of using natural language to describe the same item. In addition, this category accounts for 20.63% of all projects, making it the largest category. For a bug report, *Environment* is not only important information for reproducing the bug, but sometimes it is also one of the reasons why the bug happened.

***Screenshots (30)*:** This category mainly guides to provide screenshots to help explain the bug. There are 30 semantic-similar items with 1 expression in this category, and it accounts for 5.24% of all items. *Screenshots* is different from other categories in that it requires images rather than text, which helps to better describe the bug.

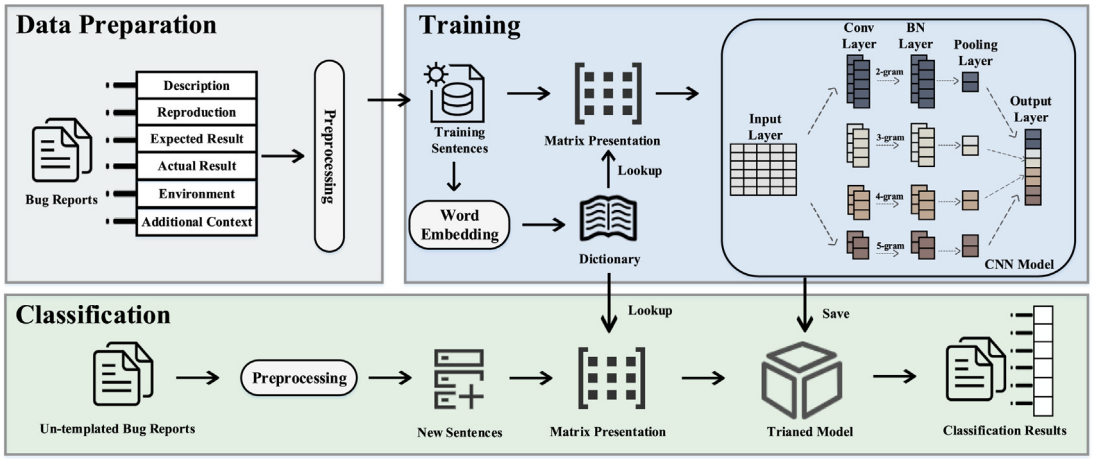


Fig. 7. Overall framework of our approach.

***Additional Context (52)*:** This category mainly guides to describe any other context about the bug. There are 52 semantic-similar items with 10 expressions in this category, and it accounts for 9.09% of all items. Interestingly, the items in the *Additional Context* category may have different meanings in different templates. This is because this category guides to describe anything other than the other items of the template it is in. For example, "What else should we know about your project/environment", since the template does not have a *Environment* item, the item also guides to report *Environment*.

***Log (15)*:** This category mainly guides to describe any error logs about the bug. There are only 15 semantic-similar items with 14 expressions in this category, and it accounts for 2.62% of all items. It is usually combined with other item categories. For example, the item *screenshots or logs*, and the item *other info/logs*.

***Solution (7)*:** This category mainly guides to provide any ideas on how to solve the bug. There are only 7 semantic-similar items with 3 expressions in this category, and it accounts for 1.22% of all items. In addition, its content may appear in the *Additional Context* of some templates.

***Checklist (5)*:** This category mainly guides to let the reporter to ensure some steps before writing. There are only 5 semantic-similar items with 4 expressions in this category, and it accounts for 0.87% of all items. Even without this category, it does not affect the understanding of the bug. For example, the item *Preflight Checklist* is meant to guide the reporter to confirm that the relevant steps listed are completed by replacing "[ ]" with "[x]" before writing the report.

***Others (38)*:** This category contains 38 (6.64%) items that cannot be classified. The several reasons are as follows: (1) There are some items defined with different meanings in different templates. For example, the explanation of the item *bug report* in the project "bvaughn/react-virtualized" is "Please include either a failing unit test or a simple repro", so the item should be in the *Reproduction* category. But its explanation in the project "SeleniumHQ/selenium" is "a clear and concise description of what the bug is", so the item should belong to the *Description* category. (2) There are some items (e.g., *affected features*) that are specific to the project. (3) There are some items (e.g., *scenario*) for which the template does not provide explanations, so items may not be classified.

As a result, in addition to the *Other* category, there are ten categories of template items. But three of these categories are not universal, namely *Log* (2.62%), *Solution* (1.22%), *Checklist* (0.87%). Therefore, we suggest a common template which is a combination of the other seven item categories: *Description*, *Reproduction*, *Expected Behavior*, *Actual Behavior*, *Environment*, *Screenshots*, *Additional Context*. The common template can guide to describe what the bug is, how the bug happened, what should have happened, what actually happened, and where it happened, relevant screenshots, and any other information about it.

From our taxonomy, we can know that the main purpose of a bug report is to help developers understand the bug and reproduce it. With the common bug report template, it not only can help to write a complete and well-structured bug report, but also can help the un-templated bug report to be templated.

> *Summary:* The common items are *expected behavior*, *describe the bug*, *to reproduce*, *additional context* and *environment* etc. We summarize a taxonomy of items with 11 categories, and suggest a common bug report template through this taxonomy.

7. RQ4: How to templatize a bug report?

We propose a deep-learning based automatic templating approach and set up experiments to evaluate its effectiveness. The automatic templating task is formalized as a sentence classification task. The input is the sentences of an un-templated bug report, the output is the predicted labels of the sentences, and the classification labels of the sentences are item categories in the bug report template.

7.1. Approach

Fig. 7 illustrates the overall framework of our approach. It contains three phases: data preparation, training, and classification. In the data preparation phase, we extract the sentences belonging to categories from bug reports. After preprocessing, we can get the training sentences. In the training phase, we first learn word embedding to generate a dictionary and convert each sentence represented by text into a matrix. Then we take the training sentences represented by matrices as input to train a CNN model. In the classification phase, for an un-templated bug report, we first divide it into sentences, preprocess the sentences, and convert the sentences to matrices. Then we feed the matrix representation of each sentence into the trained model to predict the label of each sentence.

7.1.1. Data preparation

To obtain the training sentences with classification labels, we first extract the sentences belonging to categories from bug reports, and then preprocess the sentences of each category using Lucene's StandardAnalyzer (Bialecki et al., 2012) in two steps: tokenization and removal of stop words. For tokenization, we only keep tokens that contain English letters, and we only extract English letters and numbers in a reserved token (e.g., "version:" is converted to "version"). And all words are converted to lowercase. For removal of stop words, we use the default vocabulary of English stop words (Bialecki et al., 2012). And the words are not stemmed in this experiment because the tense of the words may affect the classification results of the sentences.

7.1.2. Training

In this phase, the preprocessed sentences represented by text will be embedded into vectors and then fed into the CNN model. To convert each sentence into a matrix, we need to learn the word embedding of all unique words and generate a dictionary. After getting the matrices, we take them as input to train a CNN model to obtain the classification results.

The CNN model consists of an input layer, convolutional layer, batch normalization (BN) layer, pooling layer, and output layer. The input layer of CNN receives the matrix representation of a sentence as input. After receiving the input matrix, the convolutional layer using various filters (Ioffe and Szegedy, 2015; Wu et al., 2015; He et al., 2015; Szegedy et al., 2016) on it to extract the information. A BN layer (Ioffe and Szegedy, 2015) is added after the convolutional layer to improve training speed and model generalization performance. After the output vectors of the convolutional layer are normalized, the max-pooling is applied to get the most important feature captured by each filter. Then, the pooling outputs are concatenated and delivered to the output layer. Since each neuron in the output layer corresponds to one unique classification category (i.e., one for each item category), the output layer first performs a linear transformation on the input vector and then applies a softmax function to normalize it.

The CNN model uses the loss function to calculate the deviation of the prediction results from the ground truth labels, which is used in the backpropagation process to update the gradients. In order to obtain the best CNN model, the loss function needs to be converged by continuously training and optimizing the parameters in the CNN. Our loss function is set to:

$$Loss(P, R) = -\sum_{i}^{n} R_i log(P_i) \tag{1}$$

In the above equation, R denotes the ground truth vector and R_i denotes the i_{th} element value of R. If R belongs to class i, R_i is 1, else is 0. P represents the corresponding prediction vector, which is the normalized vector generated by the output layer. And the value n means the total number of categories.

7.1.3. Classification

In order to templatize bug reports that are not using templates, we need to classify the sentences in the bug reports. We first divide the bug reports into sentences and preprocess the sentences in the same way as the *Data Preprocessing* phase. After preprocessing, we convert each sentence to a matrix by looking up the dictionary learned in the training phase. To obtain the classification result of each sentence (i.e., the item category of the sentence), we finally feed the matrix representation of each sentence into the trained CNN.

Table 6
Distribution of the dataset.

Category	Training	Validation	Testing
Description	7312	2438	2438
Reproduction	7371	2457	2457
Expected behavior	2732	911	911
Actual behaviors	7444	2482	2482
Environment	8894	2965	2965
Additional context	3415	1139	1139

7.2. Study setup

In this section, we provide the details of our experimental setup. We first describe the experimental dataset, and then introduce the evaluation metrics and the comparisons of approach.

7.2.1. Experimental dataset

We construct a multi-project dataset to evaluate the performance of our approach. We use the item categories from the common template suggested from RQ3 as our classification labels, including *Description*, *Reproduction*, *Expected Behavior*, *Actual Behavior*, *Environment*, and *Additional Context*. *Screenshots* is not used because the content of this category is not textual content. To minimize the noise in the experimental dataset, we first reserve the projects with templates that contain at least five item categories, and filter out the bug reports with empty content under the category. Finally, the sentences of bug reports with categories are extracted to form our experimental dataset. As a result, there are 19 166 bug reports, and 61 952 sentences in our dataset.

Then we randomly divide the dataset into a training set, a validation set, and a testing set for each category in a 6:2:2 ratio. To eliminate randomness, we repeat the predictions ten times and record the average evaluation results. Table 6 shows the number of sentences in the training set, validation set, and testing set for each category in the dataset.

7.2.2. Evaluation metrics

For each sentence in the testing set, there are four possible prediction results. Therefore, we record four basic statistics for each class: TP_i (true positive for class C_i) denotes the number of sentences that are predicted as class C_i and they do belong to class C_i. FP_i (false positive for class C_i) denotes the number of sentences that are predicted as class C_i and they actually do not belong to class C_i. FN_i (false negative for class C_i) denotes the number of sentences that are not predicted as class C_i and they do belong to class C_i; TN_i (true negative for class C_i) denotes the number of sentences that are not predicted as class C_i and they truly do not belong to class C_i. Based on the above four statistics, we compute Accuracy to evaluate the overall performance of our approach, and compute Precision, Recall, and F1-score for each class to evaluate the performance for a specific item category (Rahman and Devanbu, 2013; Jiang et al., 2013; Nam et al., 2013; Guo et al., 2003; Menzies et al., 2010).

Accuracy indicates the proportion of all correctly predicted sentences to all sentences. The value n in the equation means the total number of categories.

$$A = \frac{\sum_i^n TP_i}{\sum_i^n TP_i + \sum_i^n FN_i} \tag{2}$$

Precision indicates the proportion of all sentences that are correctly predicted as class C_i to all sentences that are predicted as class C_i.

$$P_i = \frac{TP_i}{TP_i + FP_i} \tag{3}$$

Table 7
The accuracy achieved by different approaches.

Approach	Ours	RCNN	KNN	SVM	CART
Accuracy	**0.718**	0.540	0.503	0.474	0.469

Table 8
The F1-score achieved by different approaches for each item category.

Category	Ours	RCNN	KNN	SVM	CART
Description	**0.683**	0.529	0.467	0.453	0.430
Reproduction	**0.653**	0.474	0.425	0.439	0.402
Expected behavior	**0.585**	0.009	0.233	0.099	0.230
Actual behavior	**0.678**	0.473	0.475	0.392	0.464
Environment	**0.864**	0.786	0.692	0.642	0.669
Additional context	**0.732**	0.375	0.547	0.474	0.402
Avg.	**0.717**	0.492	0.504	0.459	0.471

Recall indicates the proportion of all sentences that are correctly predicted as class C_i to all sentences belong to class C_i.

$$R_i = \frac{TP_i}{TP_i + FN_i} \tag{4}$$

F1-score is a summary measure that combines precision and recall.

$$F_i = \frac{2 * P_i * R_i}{P_i + R_i} \tag{5}$$

7.2.3. Comparisons of approach

In general, the templating task belongs to text classification. We choose the following baseline approaches to compare the performance of our approach:

TextRCNN: Lai et al. (2015) proposed the recurrent convolutional neural network (RCNN) and applied it to the task of text classification. The model can infer the label or set of labels for a given text (sentence, document, etc.) and can be used to solve our problem. We compare our approach with TextRCNN based on the same dataset and the same word embedding method.

SVM, CART, KNN: Since our approach is based on text classification, we also design three simple baseline approaches that use text classification techniques. We choose Support Vector Machine (SVM), Classification and Regression Trees (CART), and k-Nearest Neighbor (KNN) to build three different classifiers, which are widely used in previous studies (Han et al., 2011; Jiang et al., 2013; Tantithamthavorn et al., 2015). Our implementation is based on scikit-learn with default settings, which is an open source python language based machine learning toolkit. Similarly, the comparisons are based on the same dataset and the same word embedding method.

7.3. Results

The accuracy results of each approach are shown in Table 7, and the best result is in bold. Our approach achieves an accuracy of 0.718. In comparison, RCNN, KNN, SVM, and CART achieve an average accuracy of 0.540, 0.503, 0.474, and 0.469, respectively. In summary, our approach improves the average accuracy over RCNN, KNN, SVM, and CART by 33%, 43%, 51%, and 53%, respectively.

Tables 8–10 respectively show the F1-score, precision, and recall achieved by each approach in each item category. Considering the category imbalance in the dataset, we present the weighted average (Sklearn Metrics, 2022) of metrics weighted by the number of sentences in each category in the dataset. All best results are highlighted in bold.

Table 8 shows the F1-score achieved by each approach in each item category. It can be seen that our approach achieves an F1-score that ranges from 0.585 to 0.864, with a weighted average of 0.717. In contrast, the average F1-score obtained by RCNN, KNN, SVM, and CART are 0.492, 0.504, 0.459, and 0.471, respectively. We see that our approach achieves a higher F1-score than any other baseline approach in each category. In conclusion, our approach improves RCNN, KNN, SVM, and CART in terms of F1-score by 46%, 42%, 56%, and 52%, respectively.

Table 9
The precision achieved by different approaches for each item category.

Category	Ours	RCNN	KNN	SVM	CART
Description	**0.647**	0.509	0.430	0.487	0.427
Reproduction	**0.649**	0.401	0.389	0.333	0.382
Expected behavior	0.636	0.800	0.281	**0.842**	0.228
Actual behavior	**0.706**	0.479	0.503	0.485	0.457
Environment	**0.849**	0.746	0.714	0.609	0.691
Additional context	0.767	0.618	0.621	**0.795**	0.452
Avg.	**0.718**	0.570	0.511	0.540	0.475

Table 10
The recall achieved by different approaches for each item category.

Category	Ours	RCNN	KNN	SVM	CART
Description	**0.724**	0.550	0.510	0.423	0.433
Reproduction	**0.657**	0.401	0.468	0.643	0.425
Expected behavior	**0.542**	0.004	0.199	0.053	0.233
Actual behavior	**0.652**	0.478	0.449	0.329	0.471
Environment	**0.879**	0.830	0.672	0.680	0.648
Additional context	**0.701**	0.269	0.488	0.338	0.362
Avg.	**0.718**	0.527	0.503	0.474	0.469

Tables 9 and 10 respectively show the precision and recall achieved by each approach in each item category. It can be seen that our approach achieves the best results for most categories, while SVM achieves the best precision for *Expected Behavior* and *Additional Context*. For the category *Additional Context*, although SVM achieves the highest precision (0.795), our approach is very close to its results (0.767). Moreover, the recall of its approach (0.338) is much lower than that of our approach (0.701). This situation may be due to the fact that SVM is insufficient to cover all the important features of *Additional Context* and therefore can only identify only a small part of the sentences of *Additional Context*. Similarly, SVM achieves the highest precision of 0.842 for *Expected Behavior* but with a relatively low recall (0.053). One reason is that only a small proportion of sentences belong to the *Expected Behavior*, which makes SVM prefer the other category, thus misclassifying most *Expected Behavior* sentences as the other categories.

> *Summary:* For each category, our approach achieves the best performance in terms of F1-score. For all categories, our approach achieves an accuracy of 0.718 and an F1-score of 0.717 on average, which improves the other baselines by a substantial margin.

8. Discussion

In this section, we discuss the evaluation of un-templated bug reports and the implications of our study.

8.1. Evaluation of un-templated bug reports

To further demonstrate the value of our proposed approach, we apply it to the real scenario of templating bug reports written without using templates.

Evaluation Methodology: Since the bug reports written without using templates need to be manually labeled, we randomly

Table 11
The evaluation results by different approaches for untemplated bug reports.

Approach	Ours	RCNN	KNN	SVM	CART
Accuracy	**0.566**	0.197	0.295	0.287	0.205
F1-score	**0.572**	0.195	0.165	0.151	0.212

select 10 bug reports containing a total of 122 sentences. We first divide the bug reports into sentences and manually label all sentences. To minimize errors, we first ask two volunteers to manually and independently label all the sentences based on six common items as *RQ3* summarized. If there is disagreement, the two volunteers read the whole bug report and discuss it to make the final decision. After manual labeling, we preprocess the sentences in the same way as the *Data Preprocessing* phase and convert each sentence into a matrix during the trained dictionary. Finally, the matrix representations of all sentences are fed into our trained model and other baselines' trained models.

Evaluation Results: The accuracy and F1-score results of each approach are shown in Table 11, and the best results are in bold. It can be seen that our approach achieves an accuracy of 0.566 and an F1-score of 0.572. In contrast, the accuracy obtained by RCNN, KNN, SVM, and CART are 0.197, 0.295, 0.287, and 0.205, respectively. And the F1-scores obtained by RCNN, KNN, SVM, and CART are 0.195, 0.165, 0.151, and 0.212, respectively. In summary, our approach achieves higher accuracy and F1-score than any other baseline approach.

8.2. Implications

The readability of bug reports is critical to facilitate coordination and communication among developers, so it is important to understand the content of bug report templates and how to templatize bug reports. We discuss the implications for developers and researchers.

Implications for Developers: Our analysis of how common templates are can make developers aware of the popularity of bug report templates. Our analysis of the benefits of using templates can make developers aware of the importance of bug report templates. And our analysis of template content can help project managers understand how to go about writing a good bug report template for a project. Our automatic templating approach can help developers templatize an un-templated bug report. In addition, if the automatic templating approach is integrated into GitHub, automated feedback can be provided to the bug reporter at reporting time, so that the bug reporter can know what is missing in his/her bug report and adds it accordingly (e.g., missing content about environments).

Implications for Researchers: We are the first to propose a study of bug report templates on GitHub, reflecting the importance of bug report templates. Our proposed automatic templating approach can effectively templatize un-templated bug reports, pushing the frontiers of research in the comprehensibility of bug reports. Therefore, we believe that our work highlights an opportunity for further research to gain more insights into bug reports.

9. Threats to validity

Threats to internal validity concerns the potential errors in the implementation of our experiments. The first threat has to do with identifying bug reports. To ensure the quality of the dataset, we only consider the issues labeled as "bug", "Bug", "BUG", although this may miss some bug reports. The second threat has to do with our programming. Although we have double-checked and thoroughly tested our code, there may still be mistakes that we failed to catch. We also make our source code available for other researchers to replicate and extend in order to lessen the impact of undetected errors in our programming. The subjectivity in the item taxonomy's construction poses another threat. To reduce this threat, we adhere to the strict card sorting process and ask two volunteers to label items independently, thus reducing personal bias in the manual labeling process.

Threats to external validity concerns the generalizability of our results. Our study concentrates on the bug reports of popular open source projects on GitHub. Future studies on more projects and more bug tracking systems (e.g., JIRA) are needed to mitigate this issue.

Threats to construct validity concerns the rationality between the treatment and the outcomes. To gain insight into the bug report templates, we conducted in-depth research in terms of its popularity, benefits, and content, as we believe these aspects are likely to provide unique insights and value for practitioners and researchers. In addition, we use accuracy, precision, recall, and F1-score to measure the effectiveness of the approach, which have been widely used by past studies (Rahman and Devanbu, 2013; Jiang et al., 2013; Nam et al., 2013; Guo et al., 2003; Menzies et al., 2010). Therefore, we believe that the threat to construct validity is not significant.

10. Conclusion and future work

To effectively ensure the quality of software, many projects use bug reports to collect and document bugs found by users. GitHub has proposed a template mechanism to enhance the comprehensibility of bug reports. However, there is no study on the use of bug report templates on GitHub.

In this paper, we first conduct an empirical study on the popularity, benefits, and content of bug report templates. For popularity, we find that the use of bug report templates has become increasingly popular over time. For benefits, we find that bug reports with templates have shorter resolution times and higher comment coverage. For content, we discover the most commonly used items, summarize the main item categories, and provide a common bug report template. Finally, we propose an automatic templating approach for templating an un-templated bug report.

In the future, to further improve the performance of our approach, we plan to consider the context of the sentences. Additionally, we plan to apply our approach to support other bug report content optimization tasks, such as duplicate bug report detection, automatic bug report title generation, and automatic bug report summary generation.

CRediT authorship contribution statement

Hongyan Li: Methodology, Writing – original draft. **Meng Yan:** Conceptualization, Supervision, Writing – review & editing. **Weifeng Sun:** Validation, Investigation, Writing – review & editing. **Xiao Liu:** Experiments, Empirical study. **Yunsong Wu:** Data curation, Visualization.

Declaration of competing interest

This work was supported in part by the National Key Research and Development Project (No. 2021YFB1714200), the Fundamental Research Funds for the Central Universities (No. 2022CDJDX-005), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2022TIAD-KPX0067), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), and the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study (No. SN-ZJU-SIAS-001).

Data availability

Data is already available on GitHub (Our Replication Package, 2023).

References

Abreu, R., Ivancic, F., Niksic, F., Ravanbakhsh, H., Viswanathan, R., 2021. Reducing time-to-fix for fuzzer bugs. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. IEEE, pp. 1126–1130. http://dx.doi.org/10.1109/ASE51524.2021.9678606.

Anvik, J., Hiew, L., Murphy, G.C., 2005. Coping with an open bug repository. In: Storey, M.D., Burke, M.G., Cheng, L., van der Hoek, A. (Eds.), Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005. ACM, pp. 35–39. http://dx.doi.org/10.1145/1117696.1117704.

Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., Zimmermann, T., 2007. Quality of bug reports in eclipse. In: Cheng, L., Orso, A., Robillard, M.P. (Eds.), Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange, ETX 2007, Montreal, Quebec, Canada, October 21, 2007. ACM, pp. 21–25. http://dx.doi.org/10.1145/1328279.1328284.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T., 2008. What makes a good bug report? In: Harrold, M.J., Murphy, G.C. (Eds.), Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008. ACM, pp. 308–318. http://dx.doi.org/10.1145/1453101.1453146.

Bialecki, A., Muir, R., Ingersoll, G., 2012. Apache lucene 4. In: Trotman, A., Clarke, C.L.A., Ounis, I., Culpepper, J.S., Cartright, M., Geva, S. (Eds.), Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval, OSIR@SIGIR 2012, Portland, Oregon, USA, 16th August 2012. University of Otago, Dunedin, New Zealand, pp. 17–24.

Breu, S., Premraj, R., Sillito, J., Zimmermann, T., 2010. Information needs in bug reports: improving cooperation between developers and users. In: Inkpen, K., Gutwin, C., Tang, J.C. (Eds.), Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010. ACM, pp. 301–310. http://dx.doi.org/10.1145/1718918.1718973.

Chen, S., Xie, X., Yin, B., Ji, Y., Chen, L., Xu, B., 2020. Stay professional and efficient: Automatically generate titles for your bug reports. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE, pp. 385–397. http://dx.doi.org/10.1145/3324884.3416538.

Datta, S., Lade, P., 2015. Will this be quick?: A case study of bug resolution times across industrial projects. In: Padmanabhuni, S., Nambiar, R., Devanbu, P.T., Ramanathan, M.K., Sureka, A. (Eds.), Proceedings of the 8th India Software Engineering Conference, ISEC 2015, Bangalore, India, February 18-20, 2015. ACM, pp. 20–29. http://dx.doi.org/10.1145/2723742.2723744.

Datta, S., Roychoudhuri, R., Majumder, S., 2021. Understanding the relation between repeat developer interactions and bug resolution times in large open source ecosystems: A multisystem study. J. Softw. Evol. Process. 33 (4), http://dx.doi.org/10.1002/smr.2317.

Datta, S., Sarkar, P., Majumder, S., 2014. Developer involvement considered harmful?: an empirical examination of android bug resolution times. In: Lanubile, F., Ali, R. (Eds.), Proceedings of the 6th International Workshop on Social Software Engineering, SSE 2014, Hong Kong, China, November 17, 2014. ACM, pp. 45–48. http://dx.doi.org/10.1145/2661685.2661686.

GitHub, 2019. The state of the octoverse. https://octoverse.github.com/2019/.

GitHub, 2021. The state of the octoverse. https://octoverse.github.com/2021/.

GitHub, 2022. GitHub documents. https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/managing-labels.

Guo, L., Cukic, B., Singh, H., 2003. Predicting fault prone modules by the Dempster-Shafer belief networks. In: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada. IEEE Computer Society, pp. 249–252. http://dx.doi.org/10.1109/ASE.2003.1240314.

Han, J., Kamber, M., Pei, J., 2011. Data Mining: Concepts and Techniques, third ed. Morgan Kaufmann, URL http://hanj.cs.illinois.edu/bk3/.

He, K., Zhang, X., Ren, S., Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In: 2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015. IEEE Computer Society, pp. 1026–1034. http://dx.doi.org/10.1109/ICCV.2015.123.

Herraiz, I., Germán, D.M., González-Barahona, J.M., Robles, G., 2008. Towards a simplification of the bug report form in eclipse. In: Hassan, A.E., Lanza, M., Godfrey, M.W. (Eds.), Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-Located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings. ACM, pp. 145–148. http://dx.doi.org/10.1145/1370750.1370786.

Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (Eds.), 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. ACM, pp. 34–43. http://dx.doi.org/10.1145/1321631.1321639.

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: Bach, F.R., Blei, D.M. (Eds.), Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. In: JMLR Workshop and Conference Proceedings, vol. 37, JMLR.org, pp. 448–456, URL http://proceedings.mlr.press/v37/ioffe15.html.

Jiang, H., Li, X., Ren, Z., Xuan, J., Jin, Z., 2019. Toward better summarizing bug reports with crowdsourcing elicited attributes. IEEE Trans. Reliab. 68 (1), 2–22. http://dx.doi.org/10.1109/TR.2018.2873427.

Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction. In: Denney, E., Bultan, T., Zeller, A. (Eds.), 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. IEEE, pp. 279–289. http://dx.doi.org/10.1109/ASE.2013.6693087.

Ko, A.J., Myers, B.A., Chau, D.H., 2006. A linguistic analysis of how people describe software problems. In: 2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2006), 4-8 September 2006, Brighton, UK. IEEE Computer Society, pp. 127–134. http://dx.doi.org/10.1109/VLHCC.2006.3.

Lai, S., Xu, L., Liu, K., Zhao, J., 2015. Recurrent convolutional neural networks for text classification. In: Bonet, B., Koenig, S. (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA. AAAI Press, pp. 2267–2273, URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9745.

Lamkanfi, A., Demeyer, S., 2013. Predicting reassignments of bug reports - An exploratory investigation. In: Cleve, A., Ricca, F., Cerioli, M. (Eds.), 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013. IEEE Computer Society, pp. 327–330. http://dx.doi.org/10.1109/CSMR.2013.42.

Li, X., Jiang, H., Liu, D., Ren, Z., Li, G., 2018. Unsupervised deep bug report summarization. In: Khomh, F., Roy, C.K., Siegmund, J. (Eds.), Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018. ACM, pp. 144–155. http://dx.doi.org/10.1145/3196321.3196326.

Lotufo, R., Malik, Z., Czarnecki, K., 2015. Modelling the 'hurried' bug report reading process to summarize bug reports. Empir. Softw. Eng. 20 (2), 516–548. http://dx.doi.org/10.1007/s10664-014-9311-2.

Mani, S., Catherine, R., Sinha, V.S., Dubey, A., 2012. AUSUM: approach for unsupervised bug report summarization. In: Tracz, W., Robillard, M.P., Bultan, T. (Eds.), 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. ACM, p. 11. http://dx.doi.org/10.1145/2393596.2393607.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.B., 2010. Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. 17 (4), 375–407. http://dx.doi.org/10.1007/s10515-010-0069-5.

Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp. 382–391. http://dx.doi.org/10.1109/ICSE.2013.6606584.

Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, H.V., Nguyen, T.N., 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (Eds.), 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011. IEEE Computer Society, pp. 263–272. http://dx.doi.org/10.1109/ASE.2011.6100062.

2023. Our replication package. https://github.com/Lhy-apple/Automatic-Templating-Approach.

Panjer, L.D., 2007. Predicting eclipse bug lifetimes. In: Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings. IEEE Computer Society, p. 29. http://dx.doi.org/10.1109/MSR.2007.25.

Rahman, F., Devanbu, P.T., 2013. How, and why, process metrics are better. In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp. 432–441. http://dx.doi.org/10.1109/ICSE.2013.6606589.

Ramirez-Mora, S.L., Oktaba, H., Gómez-Adorno, H., Sierra, G., 2021. Exploring the communication functions of comments during bug fixing in Open Source Software projects. Inf. Softw. Technol. 136, 106584. http://dx.doi.org/10.1016/j.infsof.2021.106584.

Rastkar, S., Murphy, G.C., Murray, G., 2010. Summarizing software artifacts: a case study of bug reports. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (Eds.), Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. ACM, pp. 505–514. http://dx.doi.org/10.1145/1806799.1806872.

Rastkar, S., Murphy, G.C., Murray, G., 2014. Automatic summarization of bug reports. IEEE Trans. Softw. Eng. 40 (4), 366–380. http://dx.doi.org/10.1109/TSE.2013.2297712.

2022. Sklearn metrics. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html.

Spencer, Donna, 2009. Card Sorting : Designing Usable Categories. Rosenfeld Media.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z., 2016. Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, pp. 2818–2826. http://dx.doi.org/10.1109/CVPR.2016.308.

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Ihara, A., Matsumoto, K., 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In: Bertolino, A., Canfora, G., Elbaum, S.G. (Eds.), 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. IEEE Computer Society, pp. 812–823. http://dx.doi.org/10.1109/ICSE.2015.93.

Wu, L., Xie, B., Kaiser, G.E., Passonneau, R.J., 2011. BUGMINER: software reliability analysis via data mining of bug reports. In: Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011. Knowledge Systems Institute Graduate School, pp. 95–100. http://dx.doi.org/10.7916/D8W95JCN.

Wu, R., Yan, S., Shan, Y., Dang, Q., Sun, G., 2015. Deep image: Scaling up image recognition. Comput. Sci..

Xia, X., Lo, D., Wen, M., Shihab, E., Zhou, B., 2014. An empirical study of bug report field reassignment. In: Demeyer, S., Binkley, D.W., Ricca, F. (Eds.), 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014. IEEE Computer Society, pp. 174–183. http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747167.

Xie, J., Zhou, M., Mockus, A., 2013. Impact of triage: A study of mozilla and gnome. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013. IEEE Computer Society, pp. 247–250. http://dx.doi.org/10.1109/ESEM.2013.62.

Zhang, T., Chen, J., Luo, X., Li, T., 2019. Bug reports for desktop software and mobile apps in GitHub: What's the difference? IEEE Softw. 36 (1), 63–71. http://dx.doi.org/10.1109/MS.2017.377142400.

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, pp. 1042–1051. http://dx.doi.org/10.1109/ICSE.2013.6606654.

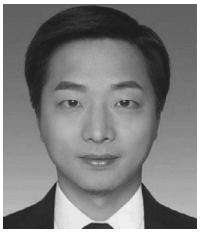
Zimmermann, T., Premraj, R., Sillito, J., Breu, S., 2009. Improving bug tracking systems. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume. IEEE, pp. 247–250. http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5070993.

Hongyan Li received the B.S. degree from Chongqing University, China, where she is currently pursuing the M.S. degree at the School of Big Data & Software Engineering. Her current research focuses on data mining and empirical software engineering.

Meng Yan received Ph.D. degree in June 2017 under the supervision of Prof. Xiaohong Zhang from Chongqing University, China. He is now a Research Professor at the School of Big Data & Software Engineering, Chongqing University, China. His current research focuses on how to improve developers' productivity, how to improve software quality, and how to reduce the effort during software development by analyzing rich software repository data.

Weifeng Sun received the B.S. and M.S. degrees from Jiangsu University, China. He is currently pursuing the Ph.D. degree at the School of Big Data & Software Engineering, Chongqing University, China. His current research interests include software testing and software debugging. His work has been published in journals and proceedings, including in IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON RELIABILITY.

Xiao Liu is currently pursuing the Ph.D. degree at the School of Big Data & Software Engineering, Chongqing University, China. His current research focuses on software engineering.



Yunsong Wu is now a Lecturer at the School of Big Data & Software Engineering, Chongqing University, China. His current research focuses on software engineering.