IMAGE RECOMMENDATION TO ENHANCE ISSUE REPORTS

XUCHEN TAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTERS OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

October 2024

Abstract

The trend of sharing images and image-based social networks has eventually changed the landscape of social networks. As a result, this shift has impacted social coding platforms, and previous studies showed that image sharing has become increasingly popular among software developers. However, most developers' productivity assistance tools predominantly rely on textual content only. To enhance issue reports, this study focuses on three primary objectives: (i) identifying issue reports that benefit from image sharing and processing in Bugzilla, (ii) identifying the type of image that would improve the bug report, and (iii) conducting a comprehensive qualitative and quantitative evaluation of the tool's performance and impact. The quantitative evaluation demonstrates that our tool achieves an average recall of 78% and an average F1-score of 74% in predicting the necessity of including image attachments in issue reports. Moreover, our qualitative evaluation of software developers showed that 75% of the developers found the overall design and recommendations of our method practically useful for issue reporting. This study, along with its associated dataset and methodology, represents the first research on recommending images to developers for enhanced issue report communication. Our results illuminate a promising trajectory for enhanced and visual productivity tools for developers.

Acknowledgements

Completing this thesis would not have been impossible without the support, guidance, and encouragement of many individuals, to whom I extend my deepest gratitude.

First and foremost, I would like to express my sincere thanks to my supervisor, Dr. Maleknaz Nayebi, for her invaluable support and selfless guidance throughout my entire master's studies. Her expertise, patience, and encouragement have played a necessary role in my academic and personal growth, for which I am profoundly grateful.

I am also deeply appreciative of the insightful feedback and suggestions provided by my thesis committee members. Their contributions are important, from the thesis to the defense exam, as well as future directions.

My heartfelt thanks go to my lab mates for their constant support and collaboration. Their dedication and passion to research have been an enduring source of inspiration. I would especially like to thank my lab mate and friend, Deenu Gengiti, whose technical expertise and experience greatly enriched my research and professional development.

I am equally grateful to our EECS department at York University for equipping me with the resources and facilities necessary to conduct my research. Their support has been instrumental in overcoming various challenges throughout my graduate studies.

Finally, I would like to express my deepest gratitude to my family and friends for their encouragement, understanding, and patience. Their belief in me has been a constant source of motivation, and I am fortunate to have them in my life.

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

List of Figures vii

List of Tables viii

1 Introduction 1

2 Related Work 5

2.1 Natural language processing on issue report 6

2.2 Visual application in Software Engineering 6

2.3 Analysis of Bugzilla Issue Reports 7

2.4 Duplicate issue report detection application 7

2.5 Text-based Bug Localization Tools 8

2.6 Duplicate Detection Tools 9

2.7 Issue Summarization Tools 9

2.8 Issue Report Generation Tools 10

2.9 Issue Report Analytical Tools 10

3 The Promise of ImageR: An Illustrative Example 12

4 Empirical Design 16

4.1 Gathering and Pre-processing Bugzilla Issues 16

4.2 Labeling Nature of Images attached to each Issue 18

5 Methodology **21**

5.1 Predicting Whether an Issue Should Include an Image (RQ1) 21

Metadata-based Classification 22

***BugzillaBERT* model** 23

Ensemble learning models 24

5.2 What Should a Recommended Image Communicate? (RQ2) 25

5.3 Qualitative Evaluation of Developers' Perceived Value of Recommendations (RQ3) 26

6 Results **29**

6.1 Predicting Whether an Issue Should Include an Image (RQ1) 29

6.1.1 **Metadata-based classification** 29

6.1.2 **$\text{BERT}_{\text{Bugzilla}}$ model** 32

6.1.3 **Ensemble learning models** 33

6.2 What Type of Images Should a Developer Add? (RQ2) 35

6.3 What is the developers' perception of the value of image recommendations for issue reports? (RQ3) 38

7 Discussion **41**

7.1 The Impact of Issue Status on **RQ1** Predictions 41

7.2 Using Simple Text Classification Techniques Instead of BERT (RQ1) . . 42

7.3 Voting Strategies (RQ1) 43

7.4 Using Issue Reports' Metadata for Predicting Image Type (RQ2) 43

8 Threats to Validity **45**

8.1 Construct Validity 45

8.2 Internal Validity 45

8.3 External Validity 46

9 Application Development **47**

9.1 Implementation 47

9.1.1 Application Project Structure 47

Client Application 48

Server Application 48

Dataset Application 48
9.2 Usage Scenario 48

10 Conclusions and Future Work **51**
10.1 Conclusions 51
10.2 Future Work 52

Bibliography **56**

List of Figures

3.1 An illustrative example of the operation of ImageR in steps 14
3.2 An effect comparison after applying ImageR recommendation 15

4.1 An Mturk task example. 18
4.2 The prediction result in RQ2. 18
4.3 The chord diagram of 10 candidate labels in RQ2. 19

5.1 The step taken to answer **RQ1** and determine whether an issue report should have images. 22
5.2 The flowchart of **RQ2** to determine what type of image should be attached to an issue report. 25
5.3 A view of the developers' task in the survey for evaluating recommendation in **RQ3**. 27
5.4 **RQ1** Feature importance using permutation on XGBoost classifier. . . 28
5.5 A scree plot of metadata attributes for feature selection in RQ1. 28

6.1 The ROC curve results of metadata classification models in **RQ1**. . . . 31
6.2 The average loss curve of *BugzillaBERT* against training epoch in **RQ2** 34
6.3 The Feature importance results of BERT. 35
6.4 The distribution of the correctly predicted labels per image in **RQ2** . . 36
6.5 The prediction distribution of image types for **RQ2** 37
6.6 The demographic and qualitative results of RQ3. 39

9.1 The application architecture process of the extension tool ImageR. . . . 49
9.2 A series of screenshots to demonstrate the operation of ImageR in use. 50

10.1 The reproduction steps of the issue report, ID: 65525, on Bugzilla. . . . 53
10.2 The flowchart of Automatic User Trace Generation Workflow. 54

List of Tables

4.1 The table of issue attributes on Bugzilla. 17

5.1 The survey questions in RQ3. 27

6.1 The performance metadata-based classification in **RQ1**. 30
6.2 The table of BERT Hyperparameter tuning results in RQ1. 33
6.3 The table of three ensemble learning models' performance in RQ1. . . 35

7.1 The table of metadata classifiers and text classifier performance in RQ1. 42
7.2 The F1-score of simple text classifiers for **RQ1** 43
7.3 The performance of ensemble learning models' with machine learning text classifiers and metadata classifiers. 43

"Seeing comes before words. The child looks and recognizes before it can speak."

John Berger, *Ways of Seeing*

Chapter 1

Introduction

Effectively identifying, reporting, and resolving issues is crucial for maintaining high-quality software products. Traditional text-based reports often fail to capture the full details or complexities of a problem, leading to misunderstandings, incorrect diagnoses, and longer resolution time [9]. To improve this process, several tools have been developed [96, 85]. Current recommendation systems and productivity tools for software development primarily rely on textual information. A substantial body of research has focused on bug triaging, resolution management, and developer recommendations [30], which is based on numerical data, such as bug metadata, or textual descriptions and/or the reproduction steps of the error [71, 97].

Developers are increasingly using images in their communications on social coding platforms [58] providing complementary context for comprehending the text of bug reports [59]. A study by Bettenburg et al. [9] involving developers from large open-source projects, including Mozilla, highlighted the benefits of using screenshots in issue reporting but also pointed out a significant lack of tool support for this practice. Subsequent studies in 2020 and 2021 by Nayebi and Adams [58, 59] showed a 200% increment in the number of images submitted by developers on Mozilla and Stack Overflow between 2013 and 2018. Research shows that including images in issue reports can greatly improve their clarity and usefulness. Images provide important additional information that complements the text in bug reports and are often essential for resolving issues [59]. Recent studies on the use of images on Stack Overflow and Mozilla [58, 59, 98] indicate that images can significantly reduce the turnaround time for bug resolution and Q/A tasks and improve understanding of the subject matter.

Our preliminary research showed that 89.9% of developers encounter issues where a visual representation would have expedited the understanding and resolution process. For instance, UI-related problems and graphical discrepancies are often difficult to describe accurately in words alone. In a survey conducted among 168 software development professionals, 86.9% reported that screenshots or diagrams helped them understand an issue that was unlikely to be solved using text-only descriptions. Furthermore, bug resolution time was reduced by an average of 30% when images were included in the reports or the questions [59].

Bugzilla, Mozilla's open-source bug repository, which has been the subject of several research studies [74], has shown an increasing number of issues that include images. Starting in 2013, Bugzilla saw an average increase of 5.6% in the number of issues with attached images per year [58]. In fact, for 67% of the issue reports in Bugzilla, images are only added after the initial response to a bug report and at the request of the community to better understand the issue. There are no structured guidelines on whether, when, and what types of images could be included to facilitate bug triaging. This lack of guidance often results in missed opportunities and longer turnaround time [59].

We argue that different types of images can be added to certain issue reports in Bugzilla to enhance clarity and facilitate the bug triaging and prioritization process. Identifying which issues would benefit from images and what types of images are most useful is the focus of this study. Our goal is to equip developers with the knowledge and tools to make their issue reports more comprehensive, facilitating quicker and more accurate resolutions. Ultimately, this will lead to better software quality and increased team productivity. Specifically, we address the following research questions (RQs):

RQ1: Which issue reports benefit most from including images?
Not every issue has and can benefit from adding images, and identifying the issues that can benefit from attaching an image is not trivial. We approach the question using supervised learning. Based on the historical issue reports we retrieved from Bugzilla between 1994-09-01 and 2022-07-19, we benchmarked with various classification model techniques using issues' metadata (e.g., bug priority, component,

severity, etc.) and the issues' textual content.

RQ2: What types of images should a developer add for an issue report?
For each issue identified that would benefit from attaching images, we recommend specific types of images that developers should add to the issue report. For this task, we relied on the ten categories of images defined in previous research [58] and offered a set of image types. We use supervised learning to answer this question. We experimented with classifiers using issues metadata and sequence-to-sequence models using the text content of the issue reports.

As a result of comprehensive benchmarking, we developed a recommendation engine and a plugin tool called *ImageR*. *ImageR* first determines whether an issue requires an image based on the findings from **RQ1**. If an image is needed, the types of images are suggested to developers for improved communication based on the results from **RQ2**. *ImageR* incorporates classic machine learning models, deep learning models, and ensemble learning techniques. We then qualitatively evaluated the methods and the tool by addressing **RQ3**:

RQ3: How do developers perceive the value of recommendations by *ImageR*?
We performed a qualitative analysis with software developers using our tool extension to report issues in Bugzilla. We surveyed 12 software developers reporting 90 bug reports on the usefulness of *ImageR* and their perceived accuracy of its predictions. We further aggregated feedback for future improvements.

The main contributions of this work include:

1. Developing the first model that incorporates classic machine learning models, deep learning models, and ensemble learning techniques to improve the bug reporting process and enhance developers' productivity using visual aids.

2. Creating a recommendation engine and plugin tool, *ImageR*, which enhances bug report communication through the complementary use of images.

3. Conducting a quantitative benchmarking of different techniques and a qualitative survey study with 12 software practitioners to demonstrate the utility of *ImageR*.

4. Compiling a dataset of historical issue reports with image attachments from *Bugzilla*.

In what follows, we discuss the design of the case study and the data used for our analysis in Chapter 4,. We then introduce the methodology and the steps taken by detailing relevant machine learning details in Chapter 5. We present our results in Chapter 6 and discuss different design options and their impact in Chapter 7. We then discuss the threats to the validity of our study in Chapter 8 followed by the related work in Chapter 2. Moreover, we discuss our ImageR extension tool in Chapter 9. We also made our dataset and evaluation results publicly available in the online supplemental material.

Chapter 2

Related Work

The empirical aspects of software development and management have primarily focused on generating actionable insights across various domains, with significant emphasis on issue tracking and triaging processes [61, 77]. Research by Bettenburg et al. [9] underscores the importance of high-quality bug reports. They found that reports lacking clear reproduction steps, expected behavior, and relevant logs significantly hinder debugging. Methods for automatically improving the quality of issue reports are a long-standing problem, especially for developers who are not experts in issue reporting. The main problem with issue reporting is the lack of experience in issue reporting that most developers have when they report issues they have. There are many studies about how to enhance the quality of issue reports [62, 60], one of them being the use of templates that guide users in providing detailed and structured information, for instance, by detailing the reproduction steps [63]. For example, software platforms like JIRA and GitHub offer issue templates that prompt users to include critical details, including *Expected Result*, *Logs*, and *Reproduction Steps* [47, 88, 64].

Several techniques and tools have been developed to automatically capture the relevant information. One notable example is the tool *Cuezilla* introduced by Zimmermann et al. [110], which measures the quality of issue reports and automatically captures and attaches relevant context information to bug reports. In their paper, they also find the most severe problems with issue reports are errors in *steps to reproduce* (S2Rs) and incomplete information, which leads the developers astray during bug resolution [110]. Further studies showed that including screenshots has a notable impact on improving bug reports [110]. Issues tracking and triaging is a

collaborative effort that calls for clear guidelines and ownership among software developers, making it a pivotal task in software teams [41, 40].

2.1 Natural language processing on issue report

Moreover, natural language processing (NLP) techniques are also commonly used to help with bug fixing [45, 95, 29]. Tarar et al. [90] introduce the automatic generation of bug report summaries, which can save time and increase efficiency during the development process. Lamkanfi et al. [44] proposed a model to predict the severity of bug reports based on other information about the bug report, particularly on the textual description. Their approach achieves a reasonable performance on the reported bug data from three open-source communities (Mozilla, Eclipse, and GNOME). Some take a different approach and mine the online forums to extract the issue reports and their reproduction steps [62, 65].

2.2 Visual application in Software Engineering

The use of visual aids in software engineering has a long and extensive history, spanning from visualizing the software development life cycle to representing algorithms for debugging [105, 4]. One of the recent uses of image analysis in software engineering focuses on software testing to detect duplicate test reports in crowd-testing platforms, leading to an increasing number of related publications [1, 106]. There are also a series of other studies on tracking the eye movement of software developers [81].

The earlier version of this study by Nayebi et al. [58] was the first to investigate the impact of image sharing on these platforms. In their study, they show that there is an increasing trend of sharing images on two popular platforms, *StackOverflow* and *Bugzilla*, and they forecast a possible future when images will become a dominating means of communication in developers' social coding environments. Moreover, visual attachments are critical in conveying an issue's context and other specifics. The study by Bissyandé et al. [11] demonstrated that bug reports accompanied by screenshots are more likely to be resolved quickly. Their research showed

that visual information helps bridge the communication gap between non-expert reporters and developers, making it easier to understand and replicate the reported issues.

2.3 Analysis of Bugzilla Issue Reports

In our study, we use Bugzilla as our main data resource. Bugzilla is an active and popular repository and community for software engineering research and software repository mining.

In recent times, the academic world has displayed a strong fascination with different facets of social coding platforms [109, 6, 86, 87], examples include contemporary version control systems like GIT and the methods and tools aimed at enhancing the productivity of software developers [56, 96, 70]. Bugzilla is an open-source platform where software maintenance requests can be grouped, implemented, and addressed by project maintainers and developers with different skills and commitment levels. [55, 36]

However, Anvik et al. [2] talk about the drawbacks of manual triaging are not able to address the increasing number of issues. In 2005, a certified maintainer from the Mozilla Software Foundation made the following comment on this situation: "Every day, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle" [2].

The reports that are posted on issue-tracking systems, such as Bugzilla or JIRA, are called bug/issue reports. How to analyze and utilize these issue report data has gained focus in recent years. Barraza et al. [5] proposed Software Reliability models and analyzed Bugzilla issue reports using models for software engineering studies. With the evolution of development tools, communication channels have increased to facilitate seamless collaboration and communication among developers [16].

2.4 Duplicate issue report detection application

Duplicate reports have been one of the biggest problems in the maintenance of issue-tracking systems, and the maintenance phase accounts for almost two-thirds of the

total effort in normal software development processes [23]. To detect and remove the duplicate reports, Wang et al. [99] proposed SETU, a duplicate crowd-testing report detection tool that combines textual descriptions and reports screenshots. Their evaluation results showed that SETU significantly outperformed other similar approaches. Cooper et al. [17] introduced a similar tool, TANGO, that integrates visual and textual information for visual bug reports, reducing 60% of developer effort under its optimal configuration. Stocco and colleagues suggested a method called VISTA for visual test repair [84]. VISTA demonstrated the ability to resolve 81% of such issues. The study also revealed that, on average, 2.8% of layout-based and 3.9% of visual test methods exhibit fragility and require maintenance [84]. Choudhary et al. introduced WebDiff for identifying cross-browser issues by visually comparing the appearance of rendered web pages [15]. This innovation paved the way for subsequent research in learning image features [80] and image-based web testing [51, 89]. Thung et al. introduces DupFinder to detect [91]

2.5 Text-based Bug Localization Tools

The bug localization which is locating the position where the bug exists in the project given an identified bug, is an indispensable process in debugging and software development [46]. Bug localization is time-consuming since the developer needs to go through all the files in a project. There are various techniques to help with automatic bug localization, mainly based on two kinds of information: historical-based or code-based. On the one hand, the bug localization tools based on historical bugs can be searching for one or more similar historical bug reports in the system since one similar bug could have been proposed and resolved before [3, 1]. Cubranic et al. [18] introduced a tool to efficiently get access to the project memory, which is a collection of software artifacts containing the past experience of the whole workgroup. Ashok et al. [3] introduced a recommender system for debugging, which can search for similar bugs from the dataset of past repositories. On the other hand, the bug localization tools based on the source code of the current project will first preprocess the source code using NLP techniques, and then find the most relevant part

in the processed source code. This technique can be considered as an information-retrieval method to localize the bug by retrieving bug-related source code. The common information-retrieval process consists of five major steps, corpus creation, indexing, query formulation, retrieval ranking, and results examination [107, 28].

Additionally, Davies et al. [21] also introduced an approach to enhancing source-code-based information retrieval by utilizing the textual similarity between bug reports. Their tool has been evaluated on 372 bugs from 4 projects and is proven to be a viable method for enhancing bug localization on all four projects [21].

2.6 Duplicate Detection Tools

Duplicate bug reports can be a significant challenge in issue-tracking systems, leading to wasted resources and confusion [78]. Tools like ReBucket and DupFinder address this by analyzing new bug reports and comparing them to existing ones in the database [20, 91]. This kind of tool uses text similarity algorithms, machine learning, and natural language processing (NLP) techniques to detect potential duplicates. For example, ReBucket clusters similar reports based on their content and provides a confidence score indicating the likelihood of duplication [20]. DupFinder leverages NLP techniques to compare the semantic meaning of bug reports, making it more effective at identifying duplicates even when the language used differs [91].

2.7 Issue Summarization Tools

Given the complexity and length of some bug reports, summarization tools help by providing concise summaries that highlight the most critical information. Many text summarization approaches exist that can be used for summary generation of bug reports by either supervised learning or unsupervised learning. Rastkar et al. [72] investigated a supervised classifier-based approach by assigning zero or one value to each sentence in the bug report conversation based on its features. Lotufo et al. [50] investigated an unsupervised approach, which ranks a bug report sentence based on whether this sentence discusses a frequently discussed topic, or based on the similarity between this sentence and the report title or description. Their paper shows a

12 percent improvement on the supervised learning approach that Rastkar et al. proposed [72]. Mani et al. [53] also explored four existing unsupervised summarization techniques for bug report summarization, and their work shows an improvement in precision over the supervised approach.

2.8 Issue Report Generation Tools

Tools that assist in the generation of issue reports often automate the process of bug finding and reporting. Yagemann et al. [103] propose Bunkerbuster, an automated data-driven bug-hunting and reporting tool. It not only speeds up the reporting process but also ensures that developers receive comprehensive and accurate information.

Additionally, debugging tools like WinDbg [1]and GDB [2] for `Windows` and `GNU` respectively, can be integrated into the issue-reporting process to capture detailed debugging information automatically, which is then included in the bug report. Platforms like JIRA and GitHub offer issue templates that prompt users to include critical details, including *Expected Result*, *Logs*, and *Reproduction Steps* [47, 88]. Using these tools can enhance the quality and detail of the reports, making it easier for developers to diagnose and fix the issues.

After an issue is reported, localization tools help in identifying the root cause of the problem. These tools often analyze the reported symptoms, log files, and system states to narrow down the source of the issue. Crash reporting tools like Google's Crashlytics and Sentry automatically capture and report crashes, often pinpointing the exact line of code where the issue occurred. Static and dynamic analysis tools such as SonarQube and Valgrind also assist in bug localization by analyzing code to detect potential issues [54, 66].

2.9 Issue Report Analytical Tools

Bettenburg et al. [9, 8] are the first to study the quality of bug reports. They have a survey on developers of Eclipse to find the information in bug reports that are

[1] `http://www.windbg.org/`
[2] `https://sourceware.org/gdb/`

used, which provides helpful information to the development of bug report analytical tools. Next year, Bettenburg et al. [9] introduced CUEZILLA to measure the quality of new bug reports and recommend which elements should be added to improve the quality.

Priority level, which is one of the most important attributes of bug report, should be assigned manually by the reporter. Tian et al. [93] introduced an automated system to assign the priority level based on multiple factors, temporal, textual, author, related-report, severity, and product.

Besides, Cunha1 et al. [19] propose a tool featuring the Vector Space Model for bug reports' analysis. They leverage information visualization techniques to facilitate the data displaying and interaction with developers during the analysis and identification of bug reports.

Additionally, Song et al. introduce BEE to automatically analyze user-written bug reports and provide feedback to reporters and developers about their bug reports using linear Support Vector Machines (SVMs)[13, 34, 82].BEE can correctly detect observed behavior, expected behavior, and the steps to reproduce in the bug report, and find the missing information [82].

Other than that, there are many other various approaches have been proposed to classify issues into different types, such as bug reports, feature requests, enhancements, questions, and even the probability of bug resolution using classification models [37, 102, 92, 108]

In summary, the integration of automated tools that capture comprehensive context, combined with advanced NLP techniques and the inclusion of visual aids, significantly enhances the quality and effectiveness of issue reports. Our work contributes to this growing body of research by specifically focusing on the automatic capture of desktop screenshots to support high-quality issue reporting.

Chapter 3

The Promise of ImageR: An Illustrative Example

A software development team works on Firefox, a widely used, multi-platform web browser. John Doe, one of the developers, encounters a frustrating issue while testing the user interface (UI) on a new version of the Firefox browser. The problem involves a wrong SVG favicon being displayed in the dark mode of Firefox, which could confuse users and lead to a poor experience. John decides to file a bug report in Bugzilla. He opens his Chrome browser, navigates to the Bugzilla bug filing page, and selects "Firefox" as the product (as shown in Figure 3.1a - step a). Next, he fills in the details: categorizing the issue as a "Defect", selecting "Theme" as the component, and specifying "Firefox 101" as the version. In the Summary field, John types "svg favicon no longer respects @media (prefers-color-scheme: dark)" and provides a detailed description of the problem (as shown in Figure 3.1b - step b). John follows by explaining the problem and pressing the "Submit" button on Bugzilla. ImageR, as a `Chrome` extension, analyzes the content and metadata of the bug report prior to final submission and provides John with a recommendation (as shown in Figure 3.1c - step c). The tool's feedback is delivered in two parts:

1. **Image Necessity:** ImageR determines that an image attachment would significantly enhance this bug report's clarity, given the UI issue's visual nature.

2. **Image Type Recommendation:** ImageR suggests that John should attach a screenshot highlighting the UI misalignment, particularly focusing on the "Code"

and "Desired Output" aspects. This recommendation is based on ImageR's understanding of similar past reports where visual evidence was crucial for quick diagnosis and resolution.

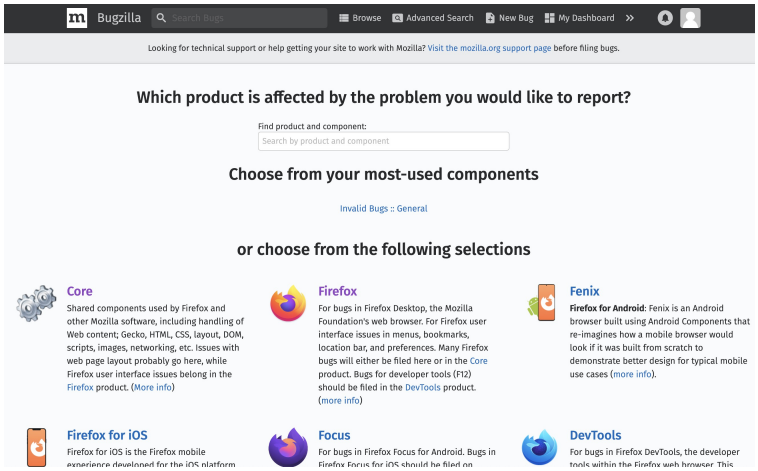
Following ImageR's recommendation, John captures a screenshot of the misaligned buttons on the Firefox Accounts page. He attaches this image to the bug report and submits it through Bugzilla.

When Tina, another developer on the team, reviews the bug report, she immediately understands the issue, thanks to the attached screenshot. The visual evidence eliminates the need for further clarification, allowing Tina to quickly identify the underlying problem and implement a fix. As a result, the bug is resolved efficiently, reducing the time from report to resolution and ensuring a smoother user experience in the next Firefox release.

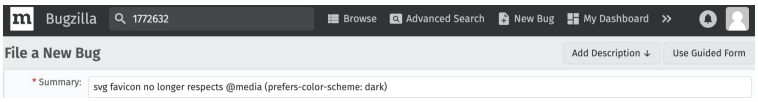
This is an issue example in reality, issue ID: 1772632 [1], which is submitted without any image by a Bugzilla user. Yet, after four posts and two days within the issues thread and the discussion, another developer, like Tina, asked for screenshots to help with the reproduction of this issue. After submitting the screenshot, the issue was understood, resolved, and closed. However, it took six days from the opening to the closing of the issue, whereas with ImageR, the immediate and automated process helps with adding images by predicting and prompting recommendations about their usefulness. Hence, we foresee a faster turnaround time for this issue.

The impact of the ImageR extension tool is also apparent, as evidenced by a comparison between issue reports with and without image attachments conforming to ImageR's recommended image types (Figure 3.2). Figure 3.2a presents the issue report of the same issue ID: 1772632, containing only plain text, without any image attachment as it was submitted in reality. In contrast, Figure 3.2b shows the enhanced version of this issue report, with an image attachment we envision can be achieved using ImageR that adheres to the image types recommended.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=1772632



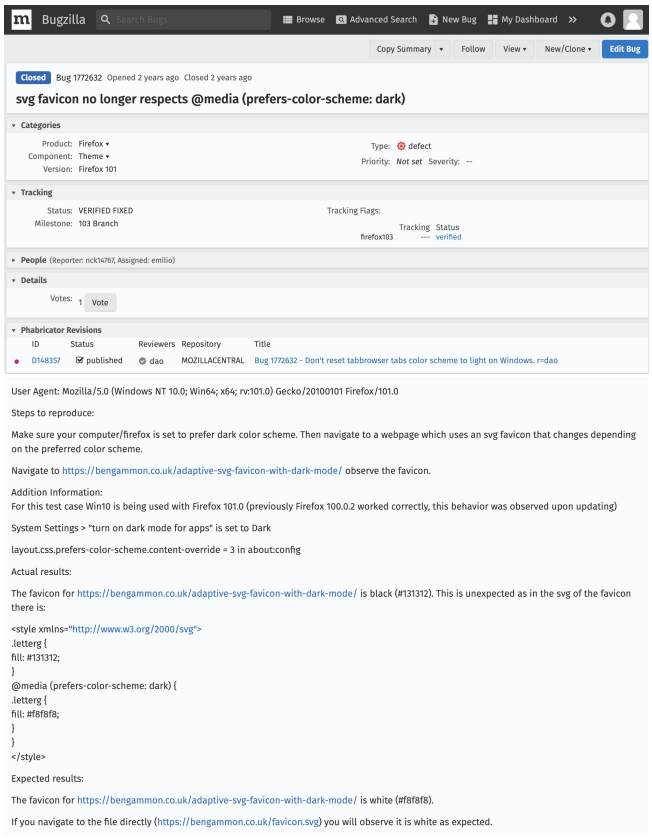
(A)



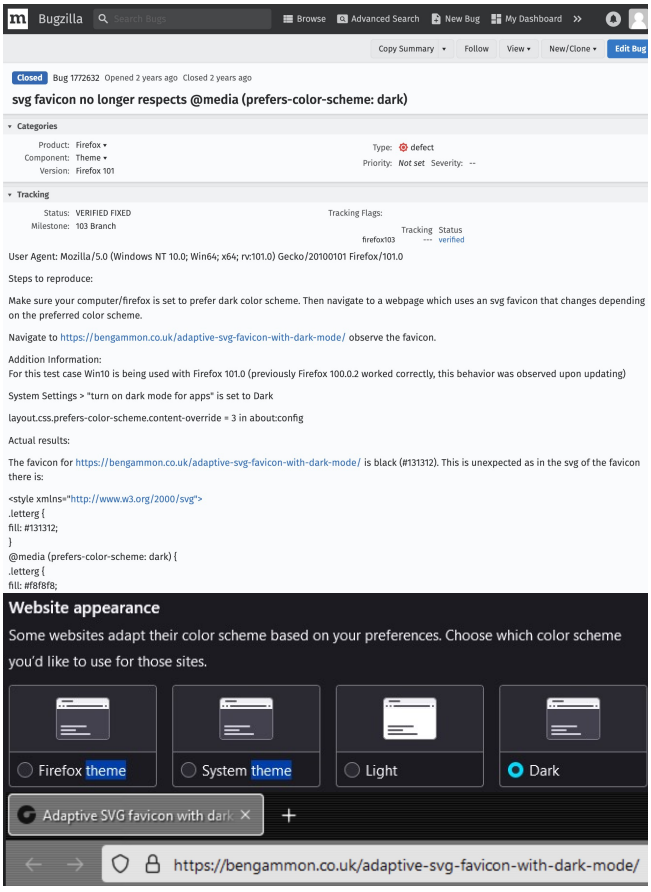
(B)

(C)

FIGURE 3.1: An example of step-by-step operation of our tool ImageR by progressive screenshots in three steps: (a) Bugzilla product selection, (b) inputting the bug attributes, and (c) ImageR recommends whether the developer should attach an image to this bug report given the input information and if so the type of the image.



(A)



(B)

FIGURE 3.2: An effect comparison between two versions of an issue report: (a) without ImageR's recommended attachment and (b) with ImageR's recommended attachment.

Chapter 4

Empirical Design

We answered our research using *Bugzilla* open-source data and followed multiple studies designed to enhance the productivity of software developers in a team.

4.1 Gathering and Pre-processing Bugzilla Issues

In the Bugzilla issue tracking system, developers can open issues, which are typically addressed or followed up by at least one other developer. The issue, along with the follow-up posts, forms an issue thread. We collected data on issues with image attachments from Bugzilla between 1994 and 2022, focusing on the twelve main products of the Mozilla organization. The issues within Bugzilla encompass various attributes and we gathered and experimented with *product*, *severity*, *priority*, *status*, *component*, *operating system*, *platform*, and *keywords* in our modeling and feature selection process in **RQ1** and **RQ2**. Furthermore, previous studies have shown that issues with images tend to have shorter lengths and are more likely to receive responses in a shorter period [59]. Hence, we also gathered data on the *number of replies* in the issue thread and the *length of the issue descriptions* as part of the issue's metadata to address our research question. Table 4.1 lists all the metadata along with their description [1].

Moreover, we used the *issue summaries* as a condensed form of information for our prediction tasks. These summaries are mandatory fields provided by developers when submitting an issue. They are manually created abstracts of the issues and are composed in natural language text. Consequently, we conducted standard text

[1] https://wiki.mozilla.org/BMO/UserGuide/BugFields

TABLE 4.1: The list of retrieved metadata information from Bugzilla. The fields defined by the developers at the time of issue submission are labeled as "Developer defined", and those attributes defined or changed later by the community engagement in the issue thread are categorized as "Bugzilla defined". The other attributes we retrieved and calculated are defined as "Researchers defined".

Field	Description	Source
Type	The type of issue, defect, task, or enhancement.	Developer defined
Summary	A few words length of the problem description.	Developer defined
Description	A detailed description of the issue.	Developer defined
Product	The software product affected by the issue.	Developer defined
Severity	A rating indicating the impact of the issue.	Bugzilla defined
Priority	A rating indicating the importance of fixing the issue.	Bugzilla defined
Status	A status indicating the current state of the issues.	Bugzilla defined
Component	The component of the software that is affected by the issue.	Developer defined
Operating System	The operating system on which the issue was observed.	Developer defined
Platform	The hardware platform on which the issue was observed.	Bugzilla defined
Number of replies	The number of responses to a submitted issue report (thread length)	Researcher calculated
Description length	The number of characters of the issue description.	Researcher calculated
Contains Image	The issue report contains image attachment or not.	Researcher calculated

pre-processing steps to prepare for our machine-learning tasks. These steps encompassed converting all words to lowercase, removing stop words using the default list from NLTK, and eliminating punctuation, noise, insignificant symbols, and emojis. Subsequently, we performed text lemmatization to map words to their root forms. We used these pre-processed summaries to address **RQ1** and **RQ2**.

For our analysis in **RQ1** and **RQ2**, we gathered 34,540 issue reports with image attachments from *Bugzilla*. This data was collected using the Bugzilla REST API. Due to the limitations of the API, we first focused on collecting issues with attachments. Yet, not all these attachments are images. Hence, after gathering issues with attachments, we determined the type of attachment file. Whenever the attachment was in the format of AVIF, JPEG, PNG, GIF, BMP, TIFF, SVG+XML, or WEBP, we classified it as

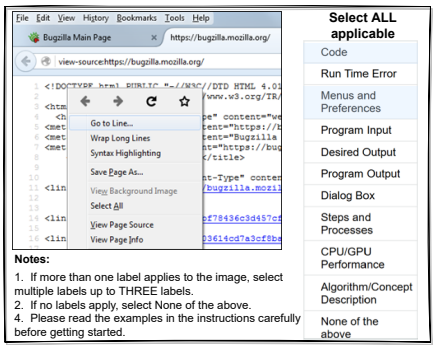


FIGURE 4.1: An example of an Amazon Mturk task presented to workers for labeling image types. The right-side menu displays a list of 10 labels from which each worker can choose up to three for the displayed image.

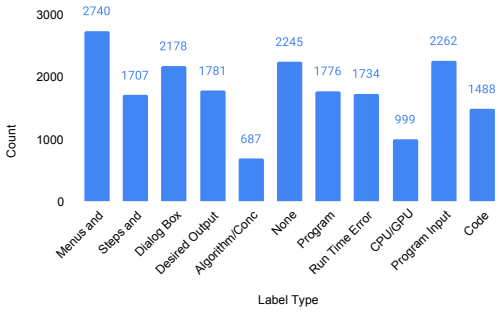


FIGURE 4.2: The histogram showing the number of correctly labeled images in **RQ2**.

an image. Consequently, we identified 17,270 issues with at least one image. These were distributed across twelve main Mozilla products, spanning from September 1^{st}, 1994, to July 19^{th}, 2022. Then, we sampled an equal number of issue reports without image attachments from Bugzilla, similarly distributed within these 28 years.

4.2 Labeling Nature of Images attached to each Issue

To answer **RQ2**, we must label the training dataset as the nature of images. We used the 10 labels introduced by Nayebi et al. [58] for categorizing the images. These labels are *Code*, *Run Time Error*, *Menus and Preference*, *Program Input*, *Desired Output*, *Program Output*, *Dialog Box*, *Steps and Processes*, *CPU/GPU performance*, *Algorithm/-Concept Description*.

For this labeling task, similar to many image processing studies [67, 83], we used `Amazon Mechanical Turk (MTurk)` [2]. `MTurk` is often used for data annotation in different contexts, including advanced image processing and captioning data sets such as the highly cited and used IMAGENET dataset [22]. Following this literature, we structured the task so three different workers labeled each image. In addition, we specifically invited "professional workers" with a proven background in "IT services and industry" who had an acceptance rate exceeding 75% on the `MTurk` platform.

[2] `https://www.mturk.com`

FIGURE 4.3: The chord diagram to show the relationship between 10 candidate labels in the dataset of RQ2.

We went a step further to control the quality of these labels manually. Afterward, one of the authors checked the labels received manually and marked any unjusti ed labels. We then discussed these with the team and adjusted the labels for any disagreements.

As a result, we labeled 30% of the images we gathered from Bugzilla. As a result, 492 distinct workers labeled 6,235 images in a way that three workers labeled each image and we adjusted 11.1% (695) of the image labels by discussing and voting within the research team.

As a result, each image has a vector of labels where for $Img(i) \in Images$ with a xed set of 10 labels, denoted as Labels The set involves the introduced set of 10 labels introduced above, $Labels = < lbl_0, lbl_2, \ldots, lbl_{10} >$, which are binary vectors to show if the label exist or not. These labels are assigned by MTurkworkers, denoted as w:

$$0 \quad \sum_{w=1}^{3} Votes_w(Img_i, lbl_j) \quad 3 \tag{4.1}$$

We represent each image with its labels in our Machine Learning models for RQ2 as a vector of 10 items, where each item can have a value greater than or equal to zero and less than four (each image can be categorized and labeled with as zero, one, two, or three category types). Figure 4.1 demonstrates an example task for a worker on Mturk, while Figure 4.2 provides an overview of the distribution of the 10 labels that were assigned by workers to our set of 6,234 images.

Figure 4.3 shows a chord diagram presenting a visual representation of the interconnected components, highlighting the interactive relationships and dependencies among them. Central to this system are the segments for Codeand Steps and Processes, which have more extensive connections, indicating their pivotal roles in the system's operation. Algorithm/Concept Description and CPU/GPU Performance show relatively signi cant interactions. The Desired Output and Program Output segments are linked to various components. Dialog Box, Menus and Preferences and Program Input represent the user interface and interaction components, which are integral for user engagement and input processing. The Run Time Error segment, while not central, is also connected to other components. Overall, the chord diagram illustrates interactions where each component plays a distinct role in the overall functionality and user experience.

Chapter 5

Methodology

We discuss the empirical methodology of our case study design on Mozilla in what follows.

5.1 Predicting Whether an Issue Should Include an Image (RQ1)

In RQ1, we aim to predict whether a given issue should include an image attachment. This task involves binary classi cation, where the prediction outcome indicates whether adding images to an issue would improve developers' communication based on historical data from Bugzilla. We used issue summaries and metadata for this classi cation, as detailed in Table 4.1. As this is the rst study in this context, we conducted a benchmark using multiple techniques to assess the performance of various methods in determining whether an image should accompany an issue. An overview of the methodology and benchmark is illustrated in Figure 5.1. To answer RQ1, we implemented three models:

1. Metadata-based Classi cation: We hypothesized that factors such as severity, priority, component, status, platform, and product type might in uence the need for an image attachment. Hence, benchmarking classi ers for driving analogy based on issues' metadata.

2. BugzillaBERT: We hypothesized that the nature of the images described in the issue text could provide reasoning clues. Therefore, we trained classi ers using the text summaries of issue reports.

FIGURE 5.1: The step taken to answer RQ1 and determine whether an issue report should have images.

3. Ensemble classification: We used ensemble methods to get the most confident prediction and benefit from the best of the above models.

In the following, we discuss each of these models in sequence.

Metadata-based Classification

We used the metadata information listed in Table 4.1 (overall 11 attributes except for issue summary) to train our models and predict whether an image attachment is needed for an issue report. The dataset comprised all the Bugzilla issues with image attachments, each with 11 features (metadata information except the issue summary) as presented in Table 4.1. First, we standardized the features using a standard scaler to scale them to unit variance. This pre-processing step ensured that all features contributed equally to the analysis. To optimize the attributes of Bugzilla issues, we performed feature selection using permutation on the XGBoost classifier [14]. Permutation feature importance in XGBoost measures the significance of each feature in a predictive model by calculating the reduction in the model's performance when the feature's values are randomly shuffled. We also verified that the dataset is linearly separable by scree plot diagram analysis in Figure 5.5.

We also applied Principal Component Analysis (PCA) for possible dimensionality reduction. PCA is commonly used to reduce the dimensionality of datasets with redundant features into a smaller set of features that captures most of the variance

in the original data [42]. We used Python's scikit-learn [1] for the PCA implementation. The number of principal components is determined by the explained variance criterion. We found the optimal number of principal components is 11 using the Elbow method, as scree plot diagram 5.5 shows.

We benchmarked with four different classifiers, namely Random Forest (RF), XGBoost (XGB), Gaussian Naive Bayes (GNB), and Support Vector Machine (SVM) for this task. We performed hyperparameter tuning with, babysitting random search, and grid search methods during training to ensure optimal performance. The performance of BERT model is highly dependent on hyperparameter tuning, and it is important to carefully select an appropriate hyperparameter tuning method to ensure optimal performance of the BERT model in specific tasks. Various techniques have been proposed for hyperparameter tuning, including the babysitting method, grid search, random search, Bayesian optimization, and population-based tuning [35, 104].

In our study, the main parameters include Learning rate (LR), Maximum Sequence Length (MSL), Epoch, Batch size, and Dropout rate. We performed benchmarking with different setups to determine the best hyperparameter setup. The tuning results are shown.

Finally, we evaluated the performance of classifiers by cross-validation and reported our results in terms of F1-score and accuracy.

BugzillaBERT model

We also implemented the BERT language model for text classification using the text summary of each issue report. BERT employs deep learning techniques to understand the contextual relationships between words. Initially pre-trained on a large corpus of text data using Masked Language Modeling (MLM), we used the pre-trained uncased BERT model, BERT-base-uncased [2], released by Huggingface [24], and fine-tuned it for our binary classification task. This model was originally trained using Wikipedia and BookCorpus data. We implemented BugzillaBERT by fine-tuning the pre-trained base model, BERT-base-uncased, with our dataset, for our specific

[1] https://scikit-learn.org/stable/
[2] https://huggingface.co/bert-base-uncased

Bugzilla issues task. This dataset comprised 34,540 issue summaries, as discussed in Chapter 4, which were pre-processed to remove special characters, emojis, URLs, and punctuation. We then tokenized the text and prepared the input in the format required by the BERT-base-uncased model.

The model was trained using the AdamW optimizer to prevent over tting and a learning rate scheduler. We employed babysitting and random search for hyperparameter tuning [33]. Babysitting involves closely monitoring the training process, including the loss and accuracy metrics, to make real-time adjustments to hyperparameters and training strategies. This method enables early detection of issues such as over tting or under tting. The key hyperparameters that signi cantly in uence the model's performance include Learning Rate (LR), Maximum Sequence Length (MSL), epochs, and batch size. The detailed speci cations of BERT model training will be provided in Chapter 6. The goal of hyperparameter tuning was to identify the optimal combination of these parameters to maximize the F1-score while minimizing the average loss. Our training was based on the BERT-base-uncased model downloaded from the google-bert repository on Huggingface, and we used scikit-learn for evaluation metrics, including accuracy and F1-score.

The model was trained on GOOGLE COLAB platform, by a NVIDIA Tesla P100-PCIe-16GB GPU card.

Ensemble learning models

To enhance the accuracy and con dence of our predictions, we employed an ensemble learning approach that combines the strengths of two models: the Metadata-based Classi er and BugzillaBERT. This combination is achieved using voting mechanisms [75]. Speci cally, we implemented and experimented with three different voting strategies [27]:

- Random voting: If the classi ers disagree on whether an issue should include an image, the ensemble model randomly selects 'Yes' or 'No' by ipping a coin.
- Negative voting: In the case of disagreement, this strategy suggests that an image is unnecessary (selecting 'No' to the RQ1 question, indicating that no image is required for the issue).

FIGURE 5.2: The flowchart of RQ2 to determine what type of image should be attached to an issue report.

- Weighted voting: When the classifiers disagree, this approach calculates the confidence for each classifier and votes for the one with higher confidence:

$$Confidence = |(P(image_{needed}), P(Image_{(not\ needed)}))| \quad (5.1)$$

We compare these strategies when reporting our results.

5.2 What Should a Recommended Image Communicate? (RQ2)

As the result of RQ1, the developer gets a recommendation on whether adding an image to their issue description would help them better communicate the issue report. RQ2 focuses on providing further suggestions of what should be included in the image. This task involved conducting a multi-label classification since an issue report could simultaneously be associated with multiple possible image types. We postulate that the issue's metadata, including factors like priority and severity, might not be relevant when deciding the image's type or content. Furthermore, we present the results of testing this hypothesis in Chapter 7.

We used the taxonomy provided by Nayebi and Adams [59] for identifying the type of image that can benefit the issues report. We used supervised learning using the data we labeled through crowdsourcing with MTurk workers (see Figure 4.2). Figure 5.2 demonstrates our methodology for RQ2.

We processed the text summaries through a standard pipeline, which included

lemmatization using the NLTKlibrary [3], as well as the removal of special characters and punctuation. Subsequently, we applied TF-IDF vectorization to tokenize the text. The data transformation phase involved creating a multidimensional matrix that linked the text with a vector of labels, denoted as $labels = < lbl_0, lbl_2, ..., lbl_{10} >$, for each image. This matrix was then formatted to suit multi-label classi cation.

There are several strategies available to perform multi-label classi cation. One common approach is problem transformation, where the problem is treated as a set of single-label classi cation tasks [94]. Speci cally, we used the problem transformation method, employing the binary relevance approach to address the multi-label classi cation task. This method involved training a binary classi er for each label using the Gaussian Naive Bayes algorithm. To mitigate class imbalance and the consequent over tting problem, we implemented data augmentation techniques to balance the training data for each label [57].

As a result of this process, the classi er generates eleven predictions for each issue. For a given issue report, each of these predictions indicates whether a particular image type is relevant or irrelevant. Each of these predictions can fall into one of four categories: True Positive (TP), True Negative (TN), False Positive (FP), or False Negative (FN) when compared to our manually labeled image dataset. To evaluate the model, we calculated the counts of TP, TN, FP, and FN, and we reported the results.

5.3 Qualitative Evaluation of Developers' Perceived Value of Recommendations (RQ3)

We employed cross-validation using Bugzilla historical data to assess our methods in RQ1 and RQ2. Additionally, we qualitatively investigated the usefulness and desirability of our method and recommendations by surveying software developers in RQ3. We asked software developers to what extent they found our recommendations for the inclusion of images and their types helpful in their communication regarding software and code issues. We collected information on the participants and their demographics, asked them to use ImageR, and solicited their feedback about

[3] https://www.nltk.org/

FIGURE 5.3: A view of the developers' task in the survey for evaluating recommendation in RQ3.

Questions	Scale
Q1. How many years of your experience in Software Engineering and Development	Years
Q2. How frequently do you share images in your daily life	1 (never) - 5 (always)
Q3. How frequently do you share images related to software development	1 (never) - 5 (always)
Q4. To what extent did you nd the recommended images for the bug report accurate?	1 (no image is needed) - 5 (precise and useful)
Q5. Having all the recommendations, how do you evaluate the overall quality of the recommendations?	1 (less useful) - 5 (highly useful)
Q6. How useful were the recommendations on whether to attach an image or not?	1 (less useful) - 5 (highly useful)
Q7. Overall, how useful were the recommendations on the types of images to amend in your practice?	1 (less useful) - 5 (highly useful)

TABLE 5.1: Survey questions used for evaluating ImageR with software developers in RQ3.

the design and the recommendations of our approach and tool. Table 5.1 shows these questions.

We recruited 12 developers to participate in our study. We assigned each developer 15 issue reports along with our predictions for RQ1 and RQ2 and presented each issue report to a developer. After they nished reading the report, we showed them the image recommendations generated by our method. Subsequently, we asked the developer about the issue report:

Figure 5.3 illustrates an example of a recommendation result shown to the developers for this evaluation task. These recommendations are for an issue report

FIGURE 5.4: RQ1 Feature importance using permutation on XGBoost classi er.

FIGURE 5.5: A scree plot of metadata attributes for feature selection. The x-axis represents the component or factor number, ordered from the most important to the least important.

with the following summary description: “Unable to log into Bugzilla with auto- ll saved long credentials from Google Chrome browser.” Our model predicted that this issue would bene t from the inclusion of an image (RQ1), and that the image should contain screenshots of the code and the Desired outcome. In RQ3, participants are asked to evaluate the precision and usefulness of these two recommendations, speci cally for the Code and Desired Output On average, each developer spent 5.2 minutes evaluating each issue report and its recommendations.

Each issue report was independently evaluated by two developers. Consequently, with 12 participants evaluating 15 issues each, we conducted a total of 90 issue evaluations for RQ3. We recruited participants using convenience sampling [39] and by advertising the survey on our LinkedIn network.

Chapter 6

Results

In what follows, we discuss the results of research questions sequentially.

6.1 Predicting Whether an Issue Should Include an Image (RQ1)

In this chapter, we present the evaluation results for all three classifiers (Section 5.1), predicting whether an issue report should include an image or not.

6.1.1 Metadata-based classification

To address RQ1 using metadata-based classification, we conducted feature selection to assess and optimize the attributes of Bugzilla issues used for training. For each Bugzilla issue report, we collected 11 metadata attributes (all attributes in Table 4.1 except the text summary). These attributes included Status, Type, Product, Component, Keywords, Operating System, Platform, Priority, Severity, Length of Description, and the Number of Replies. We also explored the potential for dimensionality reduction using Principal Component Analysis (PCA).

Results of feature selection: We initially calculated pairwise correlations between these 11 metadata attributes. Among them, the Operating System and Platform exhibited the strongest correlation, with a coefficient of 0.56. We systematically excluded one of these features at a time to assess the classification model. However, removing either the Platform or Operating System feature did not improve performance. All

TABLE 6.1: The performance metadata-based classification in RQ1.

Classifier	Precision	Recall	F1-score
SVC	0.57	0.62	0.62
Random Forest	0.67	0.74	0.74
Gaussian Naive Bayes	0.56	0.61	0.58
XGB	0.67	0.74	0.74

other correlations fell within the range of $[-0.2, 0.2]$, indicating a very weak relationship.

Furthermore, we used mean accuracy decrease to interpret the XGBoost classifier. Figure 5.4 displays the mean accuracy decrease for all features. This diagram illustrates the mean decrease in accuracy when each feature is randomly shuffled. It measures the resulting decrease in the model's performance [38]. Higher values of mean accuracy decrease indicate greater feature importance. According to Figure 5.4, Product and Component emerge as the most important features. However, it is worth noting that the differences in importance values, which fall into the range of [0.00 to 0.05], are relatively small, which is consistent with the low correlations we observed earlier.

Since our initial results were inconclusive, we performed a Scree test further to interpret the outcomes of the PCA dimension reduction approach. Figure 5.5 displays the scree plot from our PCA analysis, illustrating the relationship between the number of components and the proportion of variance explained by each component. This plot is instrumental in identifying the amount of variance accounted for by each component or factor. Typically, the scree plot is used to determine the optimal number of components by identifying an “elbow” which indicates the most significant components or features. However, as shown in Figure 5.5, our plot was anomalous, and no distinct elbow could be identified. Consequently, we did not achieve a meaningful reduction in dimensionality using PCA. Despite this, we proceeded with our analysis by training the metadata-based classifiers using all available components.

Prediction results using metadata: To determine the best-performing model for our task in RQ1, we conducted experiments using four baseline classifiers: Support

Vector Classi er (SVC), Random Forest (RF), Gaussian Naive Bayes (GNB), and eXtreme Gradient Boosting (XGBoost), The dataset used comprised of 34,540 issues discussed in Section 4.1 and we applied several pre-processing steps as discussed in Section 4.1. We performed hyperparameter tuning with babysitting [52] (manually testing a few combinations) to nd the optimal parameters of these machine learning models. Based on our analysis, we selected the following hyperparameter con gurations for training and evaluating our models: For Support Vector Classi er (SVC), we used the radial basis function (RBF) kernel, with the kernel coef cient set to 'scale' and class weights adjusted to 'balanced.' For Random Forest (RF), we chose 100 as the number of estimators with 'balanced' class weights. For Gaussian Naive Bayes (GNB), we also applied 'balanced' class weights. Finally, for XGBoost (XGB), we set 'scale_pos_weight' to 1. All other hyperparameters were left at their default values.

FIGURE 6.1: The ROC curve results of metadata classi cation models in RQ1.

The performance of the metadata classi cation models is presented in Figure 6.1 as a Receiver Operating Characteristic (ROC) curve visualization. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various classi cation thresholds for our binary classi cation task in RQ1. We employed ROC analysis to assess the classi ers' performance in addition to the classical measures. The ROC visualization demonstrates that both the Random Forest and XGBoost algorithms (almost equally) outperform the other two classi ers in classifying issues

based on metadata features. The precision, recall, and F-core performance measures for these classi ers are shown in Table 6.1.

The Random Forest and XGBoost models trained on Bugzilla issue reports demonstrates superior performance in determining whether an issue report bene ts from image attachments, excelling in both precision and recall.

6.1.2 $BERT_{Bugzilla}$ model

We utilized the BERT-base-uncasedmodel and ne-tuned it with a linear classi er trained on Bugzilla issue reports for the classi cation task of RQ1. The text of each issue report underwent preprocessing, including tokenization and cleaning as described in Sec 4.1, to prepare it for training.

Tuning results: We conducted experiments with various hyperparameters to achieve optimal performance of the BERT model [24] with babysitting [52] and grid search method [7]. Speci cally, we experimented with two, three, and four epochs. For batch size, we tested 16 and 32, observing that larger batch sizes resulted in faster training times, while smaller sizes allowed for better generalization by capturing more nuanced patterns. We also evaluated maximum sequence lengths of 128, 256, and 512. Our experiments revealed that a learning rate of 5.00E 05 consistently performed well across all epochs and batch sizes. Notably, a batch size of 64 generally outperformed the smaller sizes, while the optimal number of epochs was dependent on the combination of learning rate and batch size. The results of these tuning processes are summarized in Table 6.2. Based on this analysis, we selected the following hyperparameter combination for training and evaluating our models: batch size of 64, learning rate of 5e-05, and 4 epochs. Figure 6.3 presents the average loss and F1-score ofBugzillaBERTwith four epochs and a batch size of 64.

Prediction results using BugzillaBERT: Figure 6.2 illustrates the training progress of our BERT language model by displaying its loss function curve during the training process. The chart demonstrates the validity of our BERT model and its training, including the convergence of the model, without over tting or under tting.

This evaluation results of BugzillaBERTare consistent with the ndings in Figure 6.2. TheBugzillaBERTmodel achieved an average F1-score of 0.76, and achieved

Epoch	Batch Size	Avg loss	F1
2	64	0.391	0.774
3	64	0.248	0.761
4	16	0.632	0.635
2	16	0.639	0.619

(a) Learning Rate: 1e-4

Epoch	Batch Size	Avg loss	F1
2	64	0.437	0.765
3	64	0.380	0.767
4	16	0.232	0.759
2	16	0.411	0.767

(b) Learning Rate: 2e-5

Epoch	Batch Size	Avg loss	F1
2	64	0.420	0.776
3	64	0.342	0.771
4	16	0.188	0.760
2	64	0.402	0.774

(c) Learning Rate: 3e-5

Epoch	Batch Size	Avg loss	F1
2	64	0.400	0.780
3	64	0.303	0.767
4	16	0.175	0.758
2	16	0.394	0.778
3	32	0.258	0.772
4	64	0.156	0.760

(d) Learning Rate: 5e-5

Epoch	Batch Size	Avg loss	F1
2	64	0.575	0.718
3	64	0.553	0.719
4	16	0.504	0.746
2	16	0.411	0.767

(e) Learning Rate: 1e-6

TABLE 6.2: BERT Hyperparameter tuning results with a combination of learning rate (LR), epoch, and batch size tuning for textual classi cation model in RQ1.

an average F1-score of 0.77 when the training data. We found that the classi er achieves an average precision of 0.77 in terms of precision across all epochs and folds. Moreover, the BugzillaBERTmodel demonstrated a respectable average recall of approximately 0.74. In terms of F-score, the model achieved an average accuracy of approximately 0.77, which re ects the overall correctness of the model's predictions, indicating that it consistently made accurate classi cations across the different folds and epochs.

The BugzillaBERT model achieves a fair F1-score of 0.77 when predicting whether an issue report should be accompanied by an image or not using the issue summary.

6.1.3 Ensemble learning models

To integrate the results of our metadata-based classi er with BugzillaBERT, we developed ensemble learning methods utilizing different voting strategies. We experimented with three approaches: random selection (ipping a coin to decide), negative voting (opting for 'No' when in doubt), and weighted voting (relying on the more con dent classi er). The rationale behind these strategies is discussed in Section 5.1. Table 6.3 presents the results of combining the best-performing model from RQ1

FIGURE 6.2: The average loss curve of BugzillaBERT against training epoch in RQ2

(Random Forest) with our BugzillaBERT for each voting strategy. Among the three strategies tested, the weighted voting method, which applies Equation 5.1 to assign con dence scores to each classi er's output, achieved the highest accuracy of 0.76.

Equation 5.1 is the equation to calculate the con dence level, which is the probability [1] difference between positive and negative predictions. This compares favorably to 0.74 for random voting and 0.73 for negative voting. The weighted voting method's superior performance can be attributed to its ability to leverage the strengths of each classi er by assigning more weight to the predictions of the more con dent model. Equation 5.15.1 if it has not been explained previously, which calculates the con dence level of each classi er, was crucial in optimizing this approach.

We evaluated each strategy using precision, recall, and F1-score, nding that weighted voting consistently outperformed the others across all metrics. The nd-ings suggest that the weighted voting method effectively harnesses the complementary strengths of the Random Forest and BugzillaBERT models, resulting in improved predictive performance. Future work could explore re ning con dence calculations or incorporating additional models to enhance ensemble accuracy further.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

FIGURE 6.3: Feature importance using permutation with different learning rates on BERT in RQ1.

TABLE 6.3: The performance of ensemble learning ofBugzillaBERTand Random Forest Classi cation model using three voting strategies in response to RQ1.

Ensemble voting strategy	Precision	Recall	F1-score
Random selection	0.50	0.50	0.50
Negative voting	0.69	0.76	0.76
Weighted voting	0.66	0.78	0.74

The ensemble learning model, combining the negative voting strategy with BugzillaBERT, shows promising results in predicting whether image attachments can facilitate communication in a given issue report. The model achieves an F1-score of 78.7%, indicating its effectiveness in assisting developers in improving communications within bug reports.

6.2 What Type of Images Should a Developer Add? (RQ2)

For RQ2, we addressed a multilabel classi cation problem based on the issue's text summary to derive our conclusions. We utilized binary relevance from scikit-learn for problem transformation, enabling us to train multiple classi ers independently and optimize performance for each label. This approach allowed us to handle imbalanced classes for each label separately, addressing challenges such as skewed distributions.

The dataset is comprised of labeling data of 6,235 images by 492 distinct workers collected from Mturk , as mentioned in 4.2.

As discussed in Section 4.1 our preprocessing process includes converting all

FIGURE 6.4: The distribution of the correctly predicted labels per image in RQ2

words to lowercase, removing stop words using the default list from NLTK, eliminating punctuation, noise, insignificant symbols, and emojis, and then text lemmatization to map words to their root forms.

For each issue, we needed to make ten predictions, each corresponding to one of ten image types. Each image has ten labels to predict, and the number of correctly predicted labels for each image can range from zero to ten. We employed classification models e.g., Random Forest, SVC as introduced in Section 5.1, with hyperparameters tuned using baby-sitting method and grid search method. Evaluation metrics included precision, recall, and F1-score to provide a comprehensive assessment of performance.

Figure 6.5 shows the percentage of correct and wrong predictions for each label as the result of our cross-validation. We observed that the classifier performed better in identifying the image types of Algorithms/Concepts, which generally have distinct characteristics from the others in the text of the questions, followed by distinguishing CPU/GPU performance images, which also have distinct and often particular Windows views. However, our model struggled to differentiate between menus and dialog boxes using the text issue summaries provided in the issue reports.

Figure 6.4 displays the distribution of correctly predicted labels per image in our model. Our model correctly predicts five or more labels for 82.1% of cases and more than five labels for 66.9% of the images. On average, the model correctly predicts 6.23 labels per image, indicating its high accuracy in recognizing multiple attributes

FIGURE 6.5: The prediction distribution of image types for RQ2

in an image.

Figure 6.5 shows the percentage of True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) predictions among all predictions for the ten labels used in RQ2. TP represents correctly identified labels, TN indicates correctly rejected labels, FP refers to incorrect identifications, and FN refers to missed labels. From the diagram, we observe that for nine out of the 10 possible labels, our model correctly predicts labels (both TP and TN) more than 60% of the time. Among these image labels, the model's performance on Algorithm/Concept Description and CPU/GPU Performance is notably better, achieving an average accuracy of 68.99% and 77.29% respectively. These results demonstrate the model's effectiveness in accurately identifying key attributes, particularly those with distinct characteristics.

The significance of these findings lies in the model's ability to aid in automated image classification, potentially improving efficiency in tasks such as bug tracking and report generation. Our results form a baseline model as the first study in this direction.

A Gaussian Naive Bayes classi er trained on the text of issue summaries can predict the type of image to complement an issue report, achieving correct predictions for the majority of labels (i.e., more than ve labels) in 66.9% of the cases. This performance highlights the classi er's capability to effectively identify relevant image types based on textual descriptions, suggesting its potential utility in automating the process of enhancing issue reports with appropriate visual content.

6.3 What is the developers' perception of the value of image recommendations for issue reports? (RQ3)

We conducted a qualitative analysis with 12 developers to evaluate the effectiveness of our recommendations for 90 issue reports (Section 5.3). The developers were asked to review the recommended image type for each issue and provide feedback on whether the recommendation was appropriate. Each participant reviewed the recommendations for 15 issue reports. In this way, each issue report and its associated recommendations derived from the predictions in RQ1 and RQ2 were evaluated by two independent developers.

Among the 12 participants, six had one to three years of experience in software development, while the others had more than three. Nine of them stated that they very frequently (= 5) communicate through sharing images in their day-to-day life. This is while ten participants stated that they sometimes or more often (>= 3) post images related to their development tasks. Figure 6.6 - (a) and Figure 6.6 - (b) summarize the demographic information of our survey participants.

We presented issue reports along with our recommendations regarding whether an image should be attached and, if so, what type (both RQ1 and RQ2). Participants rated each recommendation using a ve-point Likert scale. After evaluating all 15 issue reports assigned to them, participants completed a survey about the value of images and the image recommendations they received across the reports. These results are shown in Figure 6.6 - (c) and Figure 6.6 - (d).

Our survey participants found the recommendations generated by our method to be highly useful (31.2%) or very useful (27.7%) in 58.8% of the cases. In these cases, both developers evaluating the same issue report considered the recommendations

useful. Additionally, 34.5% of the developers found at least one of the recommendations to be highly useful. Conversely, only 6.7% of the recommendations were perceived as less useful by the developers.

Even among the 6.7% perceived as less useful, both developers reached a consensus on the lack of usefulness in only 3.4% of the cases. Furthermore, when asked about the overall perceived value of the recommendations in practice, 75% of the participants rated them as useful or highly useful. Only three participants considered the overall recommendations to be relatively useful (rating them as three on our Likert scale). We could not identify any speci c factors related to their experience, such as their level of experience or tendencies in sharing images, that in uenced their ratings.

75% of the developers found our method's overall design and recommendations to be practically useful. Additionally, 93.3% of the recommendations were deemed highly useful by at least one developer.

The feedback from the developers also provided valuable insights into areas where the model could be improved, such as the need to consider additional factors beyond the issue summary when making recommendations.

"I think the model better consider more project-speci c context and the team's usual practice, and developer personal preferences to make the recommendations more spot-on."

FIGURE 6.6: (a) Years of experience in software development and the frequency of sharing images in daily life and (b) software development tasks from our survey participants. (c) Perceived need for and the type of image predictions for each of the 90 issues, and (d) the overall perceived usefulness of the 15 recommendations provided to the developer.

Furthermore, the qualitative analysis indicates that the recommendations are generally useful to the developers, as the majority of them found the recommendations helpful and relevant to their work. Some envision further usage of these results:

> "These [recommendations] could be useful for our team not only in our task management (Jira) but also in our Slack . . . I can imagine this would reduce our back-and-force questions."

Chapter 7

Discussion

We performed comprehensive experiments to discuss different possibilities and design choices for RQ1 and RQ2.

7.1 The Impact of Issue Status on RQ1 Predictions

We observed that the majority of the images were posted on Bugzilla within the discussion thread rather than in the initial issue report. Additionally, based on a literature survey involving discussions on Stack Overflow and Bugzilla, we found that in 87.8% of the surveyed posts, the inclusion of an additional image provided valuable information. Building on these observations, we formulated a hypothesis that if a developer chose to attach an image during a conversation, it was likely a correct decision, as it often added value. However, it is important to note that not all issues on Bugzilla, whether with or without images, are necessarily resolved.

Moreover, the status of an issue might affect the validity of the recommendations we provided in RQ1. We hypothesize that if an issue with an attached image remains unsolved, it suggests that adding images is of limited usefulness. We divided the dataset into subsets based on issue status to test this hypothesis and reran the evaluations. This allowed us to assess how the issues' status impacts the classification models' performance. Unsolved dataset contained issue reports in one of the statuses New, Unconfirmed, Reopened, or Assigned as defined on Bugzilla[1]. We also created the Resolved dataset containing the reports with the status Resolved, Verified, or Closed.

[1] https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Status

TABLE 7.1: The F1-score of metadata and textual classi cation models on Unsolved, Solved, and All datasets

Data	Classi er	Unsolved	Solved	All
Metadata	SVC	0.57	0.62	0.61
	Random Forest	0.71	0.76	0.74
	Naive Bayes	0.61	0.53	0.58
	XGB	0.71	0.75	0.74
BugzillaBERT	BERT	0.73	0.77	0.76

The results presented in Table 7.1 show a clear trend in the performance of the classi cation models across the different datasets. Speci cally, we observed a consistent but trivial increase in the mean F1-score for the three machine learning classi cation models (RF, SVM, XGB) trained on the unsolved, resolved, and all datasets. However, when comparing prediction performance across these subsets, the difference is minimal (2%). Therefore, we consider the impact of issue report status to be negligible in this scenario.

7.2 Using Simple Text Classi cation Techniques Instead of BERT (RQ1)

Before developing BugzillaBERT, we experimented with four traditional, non-transformer-based text classi cation techniques: Random Forest, XGB, Gaussian Naive Bayes, and SVM. We used TF-IDF for text representation and evaluated the performance of these models. Table 7.2 presents the results for issues with different statuses: Resolved, Unresolved, and All .

While we anticipated better results with BugzillaBERT through extensive training and tuning, it only achieved a slightly improved F-score of 0.77. In comparison, a simpler Stochastic Gradient Descent (SGD)[2] model achieved an F-score of 0.73. Thus, in scenarios where computational resources are limited, or explainability is required, the SGD model might also be a viable choice for RQ1. Overall, while BugzillaBERT offers superior performance in terms of F-score, the advantages of using SGD make it a valuable choice when considering the trade-offs between complexity, resource availability, and the need for model interpretability.

[2] https://scikit-learn.org/stable/modules/sgd.html

TABLE 7.2: The F1-score of simple text classifiers for RQ1

Model	Unsolved	Solved	All
TF-IDF + SGD	0.71	0.73	0.75
TF-IDF + RF	0.70	0.71	0.73
TF-IDF + MultinomialNB	0.69	0.71	0.73
BERT	0.74	0.77	0.76

7.3 Voting Strategies (RQ1)

We conducted experiments with three different voting strategies to implement our ensemble learning model in RQ1. The best performance result after comparing the three voting strategies with ensembling BugzillaBERTand Random Forest is shown in Table 6.3. The other results of the three voting strategies with other metadata classification models and text classification models are also listed in Table 7.3.

According to this comparison result, we identified the negative voting strategy as the best for creating an ensemble method of metadata-based Random Forest classifier and BugzillaBERT. The weighted voting method only performed slightly better than the standalone classifiers.

TABLE 7.3: The F1-score performance ensemble learning models of text and metadata. These results should be compared with the performance of metadata classification models in Table 7.1, and text classification models in 7.2.

F1-score	metadata:RF, text:TF-IDF&RF	metadata:XGB, text: TFIDF&RF	metadata:GNB, text:TFIDF&RF	metadata:SVM, text:TFIDF&RF
RandomSelection Ensemble	0.70	0.72	0.56	0.59
Negative Voting Ensemble	0.71	0.69	0.74	0.69
Weighted Ensemble	0.75	0.75	0.61	0.73

7.4 Using Issue Reports' Metadata for Predicting Image Type (RQ2)

In response to RQ2, we presented the results of utilizing the issue's text summary for predicting image types (see Section 6.2). To tackle this multi-label classification challenge, we applied a binary relevance transformation. We trained a Random Forest model to assess the feasibility of predicting issue types based on metadata. Similar to our approach in RQ1, we also conducted experiments using the issue's metadata for this task. We employed feature selection and explored various subsets of metadata attributes to address RQ2. However, neither the feature selection process nor the

model performance comparison showed signi cant or insigni cant features. Consequently, we used all features for the training of classi cation models, consistent with the features used in RQ1.

We achieved an F1-score of 52.4%, which was relatively low. Consequently, we shifted our focus to prediction based on issue summaries, as discussed in Section 6.2.

Chapter 8

Threats to Validity

There are multiple possibilities for the threats to the validity of this thesis. We follow the established software engineering guidelines for discussing each in this section [100].

8.1 Construct Validity

Threats to construct validity relate to the potential bias of our survey in RQ3. The survey was conducted with 12 participants with various software development experience. However, the results could still be biased and cannot be generalized to other audiences. Further evaluations of more participants with various backgrounds are needed to consolidate the evaluation while we consider the number comparable with similar studies in the eld [79].

In RQ1, while we carefully selected relevant features by feature selection approaches and tested multiple techniques, it is possible that other features could have improved the model's performance. Similarly, the hyperparameters tuning process could introduce bias to the model training. Although a systematic approach was leveraged to nd the relatively optimal hyperparameters, it is possible that other hyperparameters could have resulted in better performance.

8.2 Internal Validity

Threats to internal validity relate to evaluation metrics selection, which can also affect the model's performance. Although we used F1-score and ROC curve analysis

in our study, other metrics, such as precision and recall, may lead to different conclusions.

8.3 External Validity

Threats to external validity relate to the training data, which are historical issue reports. The quality of the dataset can impact the performance of the machine learning models. As our dataset consisted of twelve main products' issue reports extracted from Bugzilla, it might not fully represent all issue-tracking systems or software development projects. The data may be biased toward certain types of issues or projects, limiting our findings' generalizability. We mitigated this risk using balancing methods.

Finally, the status of the chosen issue reports used in the training dataset poses some threats to validity. We categorized issues according to their Bugzilla status into either *resolved* or *unresolved* datasets. In RQ1, for both machine learning and deep learning classification models, those who trained with *resolved* generally achieve higher performance compared with those trained with *unresolved*. At the same time, the performance of the models trained with *total* training data, stands in the middle place between the two groups. However, we consider this a rather trivial risk as the difference between the performance of the best classifier on each dataset was only 0.3 in the F1-Score (See Table 7.2). While selection bias and data quality can limit generalizability, the identification of relevant features and optimal hyperparameters guided future practical applications in our study on Bugzilla.

Chapter 9

Application Development

There are multiple types of tools to help with issue reports, including issue localization tools, duplicate detection tools, issue summarization tools, issue report auto-generation tools, and Issue Report Analytical Tools. For details in terms of related tools, please see Section 2.5.

9.1 Implementation

We use Python 3.8 [76] for implementing the preprocessing step as well as for the high-level scripting of the alternative solutions shown in Figure 3. The NLP pipeline and language-feature extraction is implemented using SpaCy 3.0.5 [32], NLTK 3.5 [49]. The SpanBERT-based solutions use the Transformers 4.6.1 library [101] provided by Hugging Face [1] and operated in PyTorch [68]. For the ML-based solutions, we use Scikit-learn 0.24.1 [69]. We use the Transformers library to extract embeddings. BERT's embeddings are extracted from the BERT-BASE-CASED model. The solutions and implementations proposed in this paper are implemented in Python on Google Colab[2][10].

9.1.1 Application Project Structure

The application consists of three major parts, client application, server application, and our dataset. The application architecture of the extension tool is shown in Figure 9.1, which illustrates a multi-layered architecture for ImageR, separating the tool

[1] https://huggingface.co/
[2] https://colab.research.google.com/

into four key layers: Presentation, Application, Communication, and Data. The Presentation Layer includes the user interface of the extension, responsible for the front-end interaction, using technologies like HTML, CSS, and JavaScript. The Application Layer handles core browser extension components, including manifest.json and web content as required by Google Chrome, while integrating machine learning models for processing. The Communication Layer, powered by Flask, facilitates interactions between the front-end and back-end, managing tasks like image uploads and processing requests. Finally, the Data Layer stores the issue data retrieved from Bugzilla, crowdsourcing labeled image data, and our self-annotated image attachment dataset. The models includes Random Forest, XGBoost, Support Vector Machine, and Naive Bayes models for image analysis and enhancement, and binary relevance model for multi-label classi cation.

Client Application

The client application consists of front-end components including Issue Form data Fetch Component and Recommendation Popup Component according to the requirements and standards of Chrome.

Server Application

The server application consists of back-end components including all our pre-trained classi cation models and APIs.

Dataset Application

The dataset consists of two parts, The rst part of the dataset is our dataset from Bugzilla API which is mentioned in Chapter 5, and the second part of the dataset is our data labeled from Amazon Mturk, which is mentioned in Chapter 5.

9.2 Usage Scenario

The usage scenario of this tool should be when developers nd a bug and want to le a bug-by-bug report. The tool can read the input attributes of issue reports and analyze them to decide whether or not an issue report is to be added to the issue

FIGURE 9.1: The application architecture process of the extension tool ImageR.

report. The second usage scenario should be when a developer is having an issue and wants to write an issue report to le this issue. They already know image attachment would help with the resolution of this kind of issue, however, they do not know which type of image should be attached to improve this issue report, instead of deteriorating it. The predicted recommendation would consist of a list of image attachment labels which indicate the type of image attachments, and a con dence percentage which indicates how con dent the tool is with this label prediction. With the guidance and instructions provided by recommended image labels and con dence, the user can have a clearer idea about image attachments.

In the ImageR demo, the process begins by opening a Chrome browser, navigating to Bugzilla, and selecting "Firefox" as the product for the bug report (as shown in Figure 9.2a - step a). The next step involves selecting the issue category and issue component, specifying the version, and providing a summary and detailed description (as shown in Figure 9.2b - step b). After submission, the ImageR extension tool will analyze the information of the issue (as shown in Figure 9.2c - step c). Finally, the ImageR Chrome extension analyzes the bug report, recommending to ensure its

completeness and accuracy (as shown in Figure 9.2d - step d).

To summarize, ImageR is an automated tool that generates recommendations for adding visual content to improve the bug-triaging process. It uses the issue's available metadata and textual information. Our empirical experiments show that ImageR can help increase the readability and enhance the overall effectiveness of the issue reports. Along with this tool, we also created a database of issue reports with image attachments, which can be further used in the eld.

(A)

(B)

(C)

(D)

FIGURE 9.2: An example of step-by-step operation of our tool ImageR by progressive screenshots in three steps: (a) Bugzilla product selection, (b) inputting the bug attributes, (c) ImageR tool is analyzing the issue information, and (d) ImageR recommends whether the developer should attach an image to this bug report given the input information and if so the type of the image.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

The popularity of image sharing in social networks has also impacted social coding platforms, highlighting the need for tools that can assist developers in enhancing issue reports. This thesis aimed to explore the role of images in issue reports by addressing three key research questions. First, we sought to explore when developers should add images to issue reports (RQ1). To do this, we used supervised learning with Bugzilla dataset information and sequence-to-sequence models. Second, we investigated the speci c types of images developers should use in their issue reports (RQ2). For this, we employed supervised learning, and for each issue, the report predicted whether an image type should be amended or not. Lastly, we investigated developer perceptions regarding the value of image recommendations (RQ3), by conducting a study with 12 developers evaluating 90 issue reports. The quantitative and qualitative evaluations demonstrate the usefulness of our method in recommending images. The Random Forest model outperforms other models in determining whether images should accompany an issue report (F1-score = 0.74), while the BugzillaBERTmodel achieves a fair F1-score of 0.77 in predicting the necessity of image inclusion. Furthermore, the Gaussian Naive Bayes classi er can predict the type of image to accompany an issue correctly for 66.9% of the cases. Lastly, developers have found the predictions generated by our proposed method to be highly valuable, with 75% of them expressing practical utility in the overall design and recommendations of our approach. Additionally, a signi cant 93.3% of the recommendations were considered (highly) useful evaluations by at least one

developer.

Future work can focus on enhancing the achieved accuracy, potentially by incorporating full issue report descriptions and exploring other platforms. Additionally, conducting a variety of qualitative evaluations on the visual content of social platforms is worth considering. Considering that this work is the first study in the field to recommend images for improving developers' communication, we believe this research will pave the way for further discussions on visual tools to enhance developers' productivity and promote inclusivity within our software teams.

10.2 Future Work

Addressing the research questions outlined in the thesis has provided valuable insights, several new questions are raised meanwhile.

Specifically, we have identified three key questions for future investigation:

1. What is the primary purpose of including image attachments?
2. How do image attachments facilitate issue resolution?
3. During which issue statuses are developers more likely to include image attachments?

To answer these questions, we manually reviewed 195 issue reports containing image attachments. Our analysis revealed three main purposes for including images:

1. To supplement textual descriptions by providing additional information.
2. To raise questions or concerns related to the images.
3. To respond or provide answers by attaching images.

Due to time and resource constraints, we recognize that these studies remain incomplete. However, the significance of these questions highlights the need to continue this research.

Mozilla community recommends developers include clear reproduction steps for bug reproduction in the guidance of Reporting a New Bug[1]. As an example, Figure 10.1 shows the reproduction steps of an issue report, ID: 65525 on Bugzilla[2].

FIGURE 10.1: The reproduction steps of the issue report, ID: 65525, on Bugzilla.

During this triage and resolution process, we frequently observed the requests for image-based evidence to clarify reproduction steps, particularly when there were variations in execution processes. It is known that reproducibility is critical for issue resolution[12], so we aim to develop an approach that visualizes each reproduction step of the issue report step by step. This approach will assist developers in reproducing the issue more easily, thereby improving issue resolution rates and reducing the response delay during the resolution.

Our experiment started with traditional approaches by implementing object detection and recognition models to identify software elements. Our first experiment was based on YOLOv5 model[3], fine-tuned with variety of data from popular datasets such as the Microsoft COCO dataset [48], ImageNet[22], and the Open Images Dataset [43].

However, the labels or classes of these existing datasets only cover general object classes but do not include specific classes relevant to software or computer engineering. For example, out of 80 classes of the Microsoft COCO dataset, the only classes related to computers or software are hardware objects like “Laptop”, “Mouse”, and “Keyboard” [48]. The scarcity of these instances and their variety in the datasets makes it difficult to train our models. As a result, we have identified the need to build our dataset of a scalable set of labeled images, including software, system, desktop, and other computer-related elements, which is tailored to our needs.

[1] https://www.bugzilla.org/docs/2.20/html/bugreports.html
[2] https://bugzilla.mozilla.org/show_bug.cgi?id=65525
[3] https://pytorch.org/hub/ultralytics_yolov5/

FIGURE 10.2: The implementation work ow of the automatic user trace generation process using Selenium, LLM, and online work ow development platform.

Mainstream tools for bug localization, duplicate detection, and issue summarization systems focus primarily on text-based analysis or code-level retrieval, lacking the ability to handle dynamic, context-sensitive elements [3, 18, 20, 91]. In contrast, our tool, ImageR, has shown great potential in assisting users by recommending relevant image attachments. While ImageR currently focuses on general image types, we plan to expand its capabilities to address the speci c challenges faced in software engineering, such as visualizing issue reproduction steps and capturing context-sensitive data to improve issue resolution.

A user action trace represents the sequence of actions a user performs, consisting of attributes such as user ID, action name, screenshots, action type, and other relevant metadata [31]. This trace can be modeled using action trees, which provide a hierarchical representation of user interactions. Action trees are useful for software analysis and usability evaluations, as they map out user behavior and provide a structured approach to analyzing usage patterns based on actual usage data [26, 25].

We are investigating the idea of creating an exhaustive set of screenshots by (i) identifying the set of actions users have with the browser, (ii) constructing an action tree, (iii) automating the execution of action trees, and (iv) capturing a screenshot at each step:

- To identify user actions, we began by extracting verbs from the issue reports on Bugzilla. Next, we employed a Large Language Model (LLM) to “identify all the actions a user might take when interacting with a web browser." We then prompted the model to extend this list of actions. Afterward, we asked LLM to de ne the full action trace for each of these actions. We used Dify , an

open-source LLM application development platform, to implement this work-ow. Dify supports LLM application development, from agents to complex AI work ows [4]. We choseDify for its ability to integrate LLMs with other components, facilitating the creation of an automated work ow.

- We used Selenium to simulate and automate the execution of these actions in virtual browser environment. Selenium[5] is a widely-used, portable tool for automating browsers. Selenium is commonly used for developing automated software testing frameworks for web-based applications [73].
- Finally, we developed a code snippet to capture screenshots during the execution of each action trace using Selenium. These screenshots visually represent the steps corresponding to each user action.

Figure 10.2 illustrates how this automated work ow generates user action traces based on user interactions with the software. One challenge we encountered was ensuring that all user actions were executable, which required manual debugging of the action traces. This debugging process, along with the additional runtime demands of Selenium, slowed down our overall progress. Currently, we are working on expanding these work ows and compiling a comprehensive set of screenshots for different user actions. This dataset, structured around an action tree, will support various research directions. For example, we aim to ne-tune classi cation models using these user action data. These models can be trained to classify user behavior based on a given text, such as a step in a bug reproduction process.

Code and Data Availability

The full replication repository and dataset package of this study can be found in the repository. This package is published and hosted publicly.

[4] https://github.com/langgenius/dify
[5] https://www.selenium.dev/documentation/

Bibliography

[1] Faiz Ahmed, Suprakash Datta, and Maleknaz Nayebi. “Negative Results of Image Processing for Identifying Duplicate Questions on Stack Over ow”. In: arXiv preprint arXiv:2407.05523 (2024).

[2] John Anvik, Lyndon Hiew, and Gail C Murphy. “Who should x this bug?” In: Proceedings of the 28th international conference on Software engineering. 2006, pp. 361–370.

[3] Balasubramanyan Ashok et al. “DebugAdvisor: A recommender system for debugging”. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2009, pp. 373–382.

[4] Ron Baecker, Chris DiGiano, and Aaron Marcus. “Software visualization for debugging”. In: Communications of the ACM 40.4 (1997), pp. 44–54.

[5] Nestor Ruben Barraza. “Mining bugzilla datasets with new increasing failure rate software reliability models”. In: 2017 XLIII Latin American Computer Conference (CLEI). IEEE. 2017, pp. 1–6.

[6] Andrew Begel, Robert DeLine, and Thomas Zimmermann. “Social media for software engineering”. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM. 2010, pp. 33–38.

[7] James Bergstra and Yoshua Bengio. “Random search for hyper-parameter optimization.” In: Journal of machine learning research 13.2 (2012).

[8] Nicolas Bettenburg et al. “Quality of bug reports in eclipse”. In: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange. 2007, pp. 21–25.

[9] Nicolas Bettenburg et al. “What makes a good bug report?” In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. 2008, pp. 308–318.

[10] Ekaba Bisong and Ekaba Bisong. “Google colaboratory”. In: Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners (2019), pp. 59–64.

[11] Tegawendé F Bissyandé et al. “Got issues? who cares about it? a large scale investigation of issue trackers from github”. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE). IEEE. 2013, pp. 188–197.

[12] Oscar Chaparro et al. “Assessing the quality of the steps to reproduce in bug reports”. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. 2019, pp. 86–96.

[13] Oscar Chaparro et al. “Detecting missing information in bug descriptions”. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. 2017, pp. 396–407.

[14] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016, pp. 785–794.

[15] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. “Webdiff: Automated identi cation of cross-browser issues in web applications”. In: 2010 IEEE International Conference on Software Maintenance. IEEE. 2010, pp. 1–10.

[16] Michael Chui, James Manyika, and Jacques Bughin. The social economy: Unlocking value and productivity through social technologies. Tech. rep. McKinsey Global Institute, 2012.

[17] Nathan Cooper et al. “It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports”. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE. 2021, pp. 957–969.

[18] Davor Cubranic et al. “Hipikat: A project memory for software development”. In: IEEE Transactions on Software Engineering 31.6 (2005), pp. 446–465.

[19] Carlos Eduardo Albuquerque da Cunha et al. “A Visual Bug Report Analysis and Search Tool.” In: SEKE 2010, pp. 742–747.

[20] Yingnong Dang et al. “Rebucket: A method for clustering duplicate crash reports based on call stack similarity”. In: 2012 34th International Conference on Software Engineering (ICSE) IEEE. 2012, pp. 1084–1093.

[21] Steven Davies, Marc Roper, and Murray Wood. “Using bug report similarity to enhance bug localisation”. In: 2012 19th Working Conference on Reverse Engineering IEEE. 2012, pp. 125–134.

[22] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: 2009 IEEE conference on computer vision and pattern recognition Ieee. 2009, pp. 248–255.

[23] Jayati Deshmukh et al. “Towards accurate duplicate bug retrieval using deep learning techniques”. In: 2017 IEEE International conference on software maintenance and evolution (ICSME) IEEE. 2017, pp. 115–124.

[24] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: arXiv preprint arXiv:1810.04805 (2018).

[25] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. “Mining system-user interaction traces for use case models”. In: Proceedings 10th International Workshop on Program Comprehension IEEE. 2002, pp. 21–29.

[26] Mohammad El-Ramly et al. “Modeling the system-user dialog using interaction traces”. In: Proceedings Eighth Working Conference on Reverse Engineering IEEE. 2001, pp. 208–217.

[27] Saad Ezzini et al. “Automated Handling of Anaphoric Ambiguity in Requirements: A Multi-solution Study”. In: In Proceedings of the 44th International Conference on Software Engineering (ICSE'22), Pittsburgh, PA, USA 22-27 May 2022 2022.

[28] Gregory Gay et al. “On the use of relevance feedback in IR-based concept location”. In: 2009 IEEE international conference on software maintenance. IEEE. 2009, pp. 351–360.

[29] Anjali Goyal and Neetu Sardana. “Analytical study on bug triaging practices”. In: Cognitive Analytics: Concepts, Methodologies, Tools, and Applications IGI Global, 2020, pp. 1698–1725.

[30] Tracy Hall et al. “A systematic literature review on fault prediction performance in software engineering”. In: IEEE Transactions on Software Engineering 38.6 (2011), pp. 1276–1304.

[31] Patrick Harms, Steffen Herbold, and Jens Grabowski. “Trace-based task tree generation”. In: Proceedings of the Seventh International Conference on Advances in Computer-Human Interactions (ACHI 2014). XPS-Xpert Publishing Services 2014.

[32] Matthew Honnibal, I Montani, and S Van Landeghem. “Boyd”. In: A. spaCy: industrial-strength natural language processing in Python 10 (2020).

[33] Peter Izsak, Moshe Berchansky, and Omer Levy. “How to train BERT with an academic budget”. In: arXiv preprint arXiv:2104.07705 (2021).

[34] Thorsten Joachims. “Making large-scale SVM learning practical LS-8 Report 24”. In: University of Dortmund, LS VIII-Report (1998).

[35] Tinu Theckel Joy et al. “Hyperparameter tuning for big data using Bayesian optimisation”. In: 2016 23rd International Conference on Pattern Recognition (ICPR) IEEE. 2016, pp. 2574–2579.

[36] Gladston Aparecido Junio et al. “On the bene ts of planning and grouping software maintenance requests”. In: 2011 15th European Conference on Software Maintenance and Reengineering. IEEE. 2011, pp. 55–64.

[37] Rafael Kallis et al. “Ticket tagger: Machine learning driven issue classi cation”. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE. 2019, pp. 406–409.

[38] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. “A survey of feature selection and feature extraction techniques in machine learning”. In: 2014 science and information conference. IEEE. 2014, pp. 372–378.

[39] Barbara Kitchenham and Shari Lawrence P eeger. “Principles of survey research: part 5: populations and samples”. In: ACM SIGSOFT Software Engineering Notes 27.5 (2002), pp. 17–20.

[40] Umme Ayman Koana et al. “Examining Ownership Models in Software Teams: A Systematic Literature Review and a Replication Study”. In: arXiv preprint arXiv:2405.15665 (2024).

[41] Umme Ayman Koana et al. “Ownership in the hands of accountability at brightsquid: A case study and a developer survey”. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023, pp. 2008–2019.

[42] Vipin Kumar and Sonajharia Minz. “Feature selection: a literature review”. In: SmartCR 4.3 (2014), pp. 211–229.

[43] Alina Kuznetsova et al. “The open images dataset v4: Uni ed image classi - cation, object detection, and visual relationship detection at scale”. In: International journal of computer vision 128.7 (2020), pp. 1956–1981.

[44] Ahmed Lamkan et al. “Predicting the severity of a reported bug”. In: 2010 7th IEEE working conference on mining software repositories (MSR 2010). IEEE. 2010, pp. 1–10.

[45] Dong-Gun Lee and Yeong-Seok Seo. “Systematic review of bug report processing techniques to improve software management performance”. In: Journal of Information Processing Systems 15.4 (2019), pp. 967–985.

[46] Wei Li and Ning Li. “A formal semantics for program debugging”. In: Science China Information Sciences 55 (2012), pp. 133–148.

[47] Zhixing Li et al. “To follow or not to follow: Understanding issue/pull-request templates on github”. In: IEEE Transactions on Software Engineering 49.4 (2022), pp. 2530–2544.

[48] Tsung-Yi Lin et al. “Microsoft coco: Common objects in context”. In: *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer. 2014, pp. 740–755.

[49] Edward Loper and Steven Bird. “Nltk: The natural language toolkit”. In: *arXiv preprint cs/0205028* (2002).

[50] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. “Modelling the ‘hurried’bug report reading process to summarize bug reports”. In: *Empirical Software Engineering* 20 (2015), pp. 516–548.

[51] P Lu et al. “Webpage cross-browser test from image level”. In: *2017 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE. 2017, pp. 349–354.

[52] Seyed Matin Malakouti. “Babysitting hyperparameter optimization and 10-fold-cross-validation to enhance the performance of ML methods in Predicting Wind Speed and Energy Generation”. In: *Intelligent Systems with Applications* 19 (2023), p. 200248.

[53] Senthil Mani et al. “Ausum: approach for unsupervised bug report summarization”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11.

[54] Diego Marcilio et al. “Are static analysis violations really fixed? a closer look at realistic usage of sonarqube”. In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 209–219.

[55] Humberto Marques-Neto, Gladston J Aparecido, and Marco Tulio Valente. “A quantitative approach for evaluating software maintenance services”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 1068–1073.

[56] André N Meyer et al. “Detecting developers’ task switches and types”. In: *IEEE Transactions on Software Engineering* 48.1 (2020), pp. 225–240.

[57] Agnieszka Mikołajczyk and Michał Grochowski. “Data augmentation for improving deep learning in image classification problem”. In: *2018 international interdisciplinary PhD workshop (IIPhDW)*. IEEE. 2018, pp. 117–122.

[58] Maleknaz Nayebi. “Eye of the mind: Image processing for social coding”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE. 2020, pp. 49–52.

[59] Maleknaz Nayebi and Bram Adams. “Image-based communication on social coding platforms”. In: *Journal of Software: Evolution and Process* (2023), e2609.

[60] Maleknaz Nayebi, Guenther Ruhe, and Thomas Zimmermann. “Mining treatment-outcome constructs from sequential software engineering data”. In: *IEEE Transactions on Software Engineering* 47.2 (2019), pp. 393–411.

[61] Maleknaz Nayebi et al. “Analytics for Software Project Management–Where are We and Where do We Go?” In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE. 2015, pp. 18–21.

[62] Maleknaz Nayebi et al. “Anatomy of functionality deletion: an exploratory study on mobile apps”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 243–253.

[63] Maleknaz Nayebi et al. “ESSMArT way to manage customer requests”. In: *Empirical Software Engineering* 24 (2019), pp. 3755–3789.

[64] Maleknaz Nayebi et al. “Hybrid labels are the new measure!” In: *IEEE Software* 35.1 (2017), pp. 54–57.

[65] Maleknaz Nayebi et al. “Recommending and release planning of user-driven functionality deletion for mobile apps”. In: *Requirements Engineering* (2024), pp. 1–22.

[66] Nicholas Nethercote and Julian Seward. “Valgrind: A program supervision framework”. In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66.

[67] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. “Running experiments on amazon mechanical turk”. In: *Judgment and Decision making* 5.5 (2010), pp. 411–419.

[68] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[69] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[70] Sida Peng et al. “The impact of ai on developer productivity: Evidence from github copilot”. In: *arXiv preprint arXiv:2302.06590* (2023).

[71] Danijel Radjenović et al. “Software fault prediction metrics: A systematic literature review”. In: *Information and software technology* 55.8 (2013), pp. 1397–1418.

[72] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. “Automatic summarization of bug reports”. In: *IEEE Transactions on Software Engineering* 40.4 (2014), pp. 366–380.

[73] Rosnisa Abdull Razak and Fairul Rizal Fahrurazi. “Agile testing with Selenium”. In: *2011 Malaysian Conference in Software Engineering*. IEEE. 2011, pp. 217–219.

[74] Christian Robottom Reis and Renata Pontin de Mattos Fortes. “An overview of the software engineering process and tools in the Mozilla project”. In: *Proceedings of the Open Source Software Development Workshop*. figure 1. 2002, pp. 1–21.

[75] Lior Rokach. “Ensemble-based classifiers”. In: *Artificial intelligence review* 33.1 (2010), pp. 1–39.

[76] Guido van Rossum and Fred L Drake. “Python 3 reference manual:(python documentation manual part 2)”. In: *CreateSpace, f* 121 (2009), p. 242.

[77] Günther Ruhe and Maleknaz Nayebi. “What counts is decisions, not numbers—toward an analytics design sheet”. In: *Perspectives on Data Science for Software Engineering*. Elsevier, 2016, pp. 111–114.

[78] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. “Detection of duplicate defect reports using natural language processing”. In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 499–510.

[79] Sk Golam Saroar and Maleknaz Nayebi. “Developers' perception of GitHub Actions: A survey analysis”. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 2023, pp. 121–130.

[80] Nataliia Semenenko, Marlon Dumas, and Tõnis Saar. "Browserbite: Accurate cross-browser testing via machine learning over image features". In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 528–531.

[81] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. "A systematic literature review on the usage of eye-tracking in software engineering". In: *Information and Software Technology* 67 (2015), pp. 79–107.

[82] Yang Song and Oscar Chaparro. "Bee: A tool for structuring and analyzing bug reports". In: *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2020, pp. 1551–1555.

[83] Alexander Sorokin and David Forsyth. "Utility data annotation with amazon mechanical turk". In: *2008 IEEE computer society conference on computer vision and pattern recognition workshops*. IEEE. 2008, pp. 1–8.

[84] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. "Visual web test repair". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 503–514.

[85] Margaret-Anne Storey, Brian Houck, and Thomas Zimmermann. "How developers and managers define and trade productivity for quality". In: *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*. 2022, pp. 26–35.

[86] Margaret-Anne Storey et al. "The impact of social media on software engineering practices and tools". In: *workshop on Future of software engineering research*. ACM. 2010, pp. 359–364.

[87] Margaret-Anne Storey et al. "Towards a theory of software developer job satisfaction and perceived productivity". In: *IEEE Transactions on Software Engineering* 47.10 (2019), pp. 2125–2142.

[88] Emre Sülün. "An empirical analysis of issue templates on GitHub". PhD thesis. bilkent university, 2023.

[89] Haruto Tanno et al. “Region-based Detection of Essential Differences in Image-based Visual Regression Testing”. In: *Journal of Information Processing* 28 (2020), pp. 268–278.

[90] M Irtaza Nawaz Tarar, Faizan Ahmed, and Wasi Haider Butt. “Automated Summarization of Bug Reports to speed-up software development/maintenance process by using Natural Language Processing (NLP)”. In: *2020 15th International Conference on Computer Science & Education (ICCSE)*. IEEE. 2020, pp. 483–488.

[91] Ferdian Thung, Pavneet Singh Kochhar, and David Lo. “Dupfinder: integrated tool support for duplicate bug report detection”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 871–874.

[92] Ferdian Thung, David Lo, and Lingxiao Jiang. “Automatic defect categorization”. In: *2012 19th working conference on reverse engineering*. IEEE. 2012, pp. 205–214.

[93] Yuan Tian et al. “Automated prediction of bug report priority using multi-factor analysis”. In: *Empirical Software Engineering* 20 (2015), pp. 1354–1383.

[94] Grigorios Tsoumakas and Ioannis Katakis. “Multi-label classification: An overview”. In: *International Journal of Data Warehousing and Mining (IJDWM)* 3.3 (2007), pp. 1–13.

[95] Jamal Uddin et al. “A survey on bug prioritization”. In: *Artificial Intelligence Review* 47 (2017), pp. 145–180.

[96] Bogdan Vasilescu et al. “Quality and productivity outcomes relating to continuous integration in GitHub”. In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 805–816.

[97] Romi Satria Wahono. “A systematic literature review of software defect prediction”. In: *Journal of software engineering* 1.1 (2015), pp. 1–16.

[98] Dong Wang et al. “Understanding the Role of Images on Stack Overflow”. In: *arXiv preprint arXiv:2303.15684* (2023).

[99] Junjie Wang et al. “Images don’t lie: Duplicate crowdtesting reports detection with screenshot information”. In: *Information and Software Technology* 110 (2019), pp. 139–155.

[100] Claes Wohlin et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[101] Thomas Wolf et al. “Transformers: State-of-the-art natural language processing”. In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.

[102] Bowen Xu et al. “Efspredictor: Predicting configuration bugs with ensemble feature selection”. In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2015, pp. 206–213.

[103] Carter Yagemann et al. “Automated bug hunting with data-driven symbolic root cause analysis”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 320–336.

[104] Li Yang and Abdallah Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. In: *Neurocomputing* 415 (2020), pp. 295–316.

[105] Kang Zhang, Jun Kong, and Jiannong Cao. “Visual software engineering”. In: *Wiley Encyclopedia of Computer Science and Engineering* (2007).

[106] Ting Zhang et al. “Duplicate bug report detection: How far are we?” In: *ACM Transactions on Software Engineering and Methodology* 32.4 (2023), pp. 1–32.

[107] Jian Zhou, Hongyu Zhang, and David Lo. “Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports”. In: *2012 34th International conference on software engineering (ICSE)*. IEEE. 2012, pp. 14–24.

[108] Yu Zhou et al. “Combining text mining and data mining for bug report classification”. In: *Journal of Software: Evolution and Process* 28.3 (2016), pp. 150–176.

[109] Thomas Zimmermann et al. “Mining version histories to guide software changes”. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 429–445.

[110] Thomas Zimmermann et al. “What makes a good bug report?” In: *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 618–643.