DETECTING PLANNING CONVERSATIONS IN BUG REPORTS

RAFAT BIN ISLAM
Bachelor of Science, American International University of Bangladesh, 2018

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Rafat Bin Islam, 2022

DETECTING PLANNING CONVERSATIONS IN BUG REPORTS

RAFAT BIN ISLAM

Date of Defence: December 14, 2022

Dr. John Anvik Thesis Supervisor	Associate Professor	Ph.D.
Dr. Yllias Chali Thesis Examination Committee Member	Professor	Ph.D.
Dr. Wendy Osborn Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

*To **my parents** for all their support and sacrifices*

Abstract

Software developers refer to bug reports as a reliable source of information. However, these bug reports are written in the form of conversations among developers and often become long depending on the complexity of the issue, necessitating a significant amount of time and effort to locate the desired information. Prior work focused on tagging the different types of information in the bug reports. However, their work did not identify `Plans`. In our work, we focus on retrieving `Plans` from bug reports and labeling them with a `Plan Labeller`. First, we analyzed bug reports to identify which section contains `Plans`. Then we examined three methods to detect `Plans`. Based on that, we found keywords and key-phrases to be the best approach. We applied lists of keywords and key-phrases iteratively to randomly selected bug reports to construct a list of keywords and key-phrases that can identify `Plans` in a bug report.

Acknowledgments

First and foremost, I would like to thank the Almighty Allah for giving me the opportunity to undertake this research and for His countless blessings.

I want to express my sincere gratitude to my supervisor, Dr. John Anvik. The way he treated me made me feel like he was not only my supervisor but also my guardian. Whenever I ran into any trouble or had questions about my research, he was always there to help me and guide me. I am very grateful to my supervisory committee members, Dr. Wendy Osborn and Dr. Yllias Chali, for their valuable feedback and time.

I want to thank my family for all their support and encouragement throughout my years of study. I am very grateful to my sister and brother-in-law for their countless support and guidance during my stay in Canada.

Lastly, I also want to thank the members of the Sibyl Lab for their encouragement and thoughtful comments on my research work.

Contents

Dedication **iii**

Abstract **iv**

Acknowledgments **v**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**
1.1 Contribution of This Work 4
1.2 Organization 4

2 Related Work **5**
2.1 Bug Report Summarization 5
2.1.1 Tagged Based Approaches 6
2.2 Information Extraction From Bug Reports 6
2.2.1 Pattern Based Approaches 6
2.2.2 Key-phrase Based Approaches 8
2.2.3 Keyword Based Approaches 9
2.3 Summary 11

3 Creating A Plan Labeller **12**
3.1 Plan Labeller 12
3.2 Plan Detection Process 13
3.2.1 Location of Plans in Bug Reports 14
3.2.2 Detecting Sentences with Plan 16
3.2.3 Categorization 16
3.3 Plan Identification Refinement 17
3.4 Summary 18

4 Results and Evaluation **19**
4.1 Data source 19
4.1.1 Rastkar's Dataset 20
4.1.2 The MSR Dataset 20
4.2 Bug Report Sections With Plans 23
4.3 Plan Identification 24

4.3.1 Patterns 24
4.3.2 Key-phrases 25
4.3.3 Keywords 25
4.3.4 Summary 26
4.4 Refining The Plan Detection Approach 27
4.4.1 Evaluation Metrics 27
4.4.2 First Iteration Results 28
4.4.3 Second Iteration Results 29
4.4.4 Third Iteration Results 31
4.4.5 Discussion 32
4.5 Keywords And Key-phrases Categorization 33
4.6 Final List of Keywords and Key-phrases 35
4.7 Threats to Validity 35
4.8 Summary 36

5 Implementation of the Plan Labeller **38**
5.1 An Automated Approach 38
5.1.1 Applying The Automated Approach 39
5.1.2 Results of Applying The Automated Approach 40
5.2 Discussion 40
5.3 Summary 41

6 Conclusion **42**
6.1 Future Work 43

Bibliography **44**

A List of all Keywords **48**

List of Tables

4.1 Commonly Used Keywords and Key-phrases 26
4.2 Precision, Recall and F1 Score from the iterations 33
4.3 Keyword And Key-phrase Categories 34
4.4 Final List of Keywords And Key-phrases With Category 36

5.1 Precision, Recall and F1 Score of the Automated and Manual Approach . . 41

List of Figures

3.1 Defining Plans 13
3.2 Proposed Approach. 14
3.3 Plan detection Process 15

4.1 General format of the bug reports captured from Galappaththi et al's [11] work. 21
4.2 A sample from the Rastkar et al.'s [30, 31] dataset 22
4.3 An example of `Plan` in bug description. 24
4.4 An example of `Plan` in comments. 25
4.5 Detecting keywords from the bug comments 27
4.6 The results of first Iteration 28
4.7 The results of second Iteration 29
4.8 The results of third Iteration 31
4.9 Number of Keywords And Key-phrases Used Each Iteration 35

5.1 A labeled bug report by the Plan Labeller 39
5.2 The result of automated approach 40

Chapter 1

Introduction

Bug reports are frequently consulted software artifacts because they contain valuable information such as a detailed description of the bug and a list of developer comments discussing how to reproduce or repair the bug. Developers and users refer to the bug reports for indications of whether a similar issue has been fixed in the past, to determine if the report is a duplication of prior issues, or to gather the information necessary to reproduce the bug. These bug reports are stored in a issue tracking system, where developers' and users' reports refer to both submitting unexpected behavior of software and requesting additional functionality. This issue tracking system serves as a means of communication between users and developers, allowing them to discuss potential ways of addressing issues or finding a solution.

However, bug reports are not designed to be easily readable [24]. Depending on the complexity of the bug report, the quantity of information and size of the report vary [2]. Some are small and straightforward, including a few pieces of information in a handful of sentences. While others are lengthy and complex, contain a wide variety of information, including long conversations between developers and users. Similar to posts on social media [19], bug reports also include discussion sections where developers and users can post their contributions as comments. These comments consist of numerous paragraphs of informal conversation and opinions about a problem that needs to be fixed or addressed, questions regarding the issue, and even attachments such as patches or screenshots. After each new comment, there is a chance that the conversation will become more complicated,

leaving the readers to keep track of all the new contexts on their own. Therefore, each bug report might end up with tens of comments and thousands of sentences. Keeping track of each comment and finding the desired information from those comments is not easy, and, in general, it consumes a substantial amount of time.

Given the time-consuming nature of reading and comprehending full bug reports, bug report summaries are a newly emerging area of study that might greatly benefit many developers in speeding up the bug-fixing process. In an effort to find solutions to these issues, a number of research studies [16, 21, 22, 24, 30] have presented methodologies and strategies for summarizing bug reports. On the basis of the method used to generate summary statements [31], text summarization can be characterized as either abstractive or extractive. Abstract summarizing is a technique for creating summaries based on the meaning of the original text, whereas extractive summarizing is a technique for generating summaries by extracting a paragraph or sentence from the original text. These strategies can be categorized as either supervised or unsupervised learning.

Despite the fact that these bug summarizing strategies assisted developers and users in comprehending the core topic of the bug report without reading the entire report and in locating the needed information with the least effort, they still had certain downsides. For instance, supervised learning-based techniques rely significantly on the training dataset, necessitating a significant amount of manual labor to create the annotated training dataset. As a result, several researchers have concentrated on developing unsupervised learning-based bug report summaries to minimize the need for labor-intensive manual labeling of data. Unsupervised techniques, on the other hand, are sensitive to word frequency [21] and are prone to including repetitive sentences representing a similar topic in the summary, resulting in biased findings. In addition to this, one of the most significant problems with bug summarization was that older methods frequently missed information that was either essential or valuable, and the summaries that were created by these methods did not provide any kind of user interaction that would allow users to view customized summaries based on

the information requirements that they had.

To deal with previous research issues of creating fixed-size summaries and the lack of user interaction to create bug summaries according to desired information, Galappaththi et al.[11] introduced a tag-based approach to automatically annotate the contents of a bug report. Such an approach can be used to provide an interface which will allow bug report users to find and interact with the bug report to meet their task-specific information needs without restricting the summaries to a fixed size. However, this summarization approach was found to need some improvements in delivering more accurate automated summaries to bug report users. Due to the fact that different users are interested in distinct types of information, a set of labelling modules was developed to discern the content or intent of comments in bug report comments. Among those sets of labelling modules, the `Plan Labeller` was one of the labelling modules that they introduced. However, they identified the `Plan Labeller` as part of their future work and elected not to incorporate this module into their work.

In this thesis, we focus on implementing the `Plan Labeller` for bug reports, which will help developers and users to create tag-based summaries. To achieve our goal, we identify sentences from bug reports containing `Plans` (also known as `suggestions` or `solutions`) and label those sentences with the `Plan Labeller.` Then we will examine the bug reports to determine which sections include `Plans`, and we investigate the use of three strategies based on discourse patterns, key-phrases, and keywords for identifying Plans in bug report sentences. Our work seeks to answer the following research questions:

- RQ1: Which part of the bug report contains `Plan`?

  This research question aims to understand whether `Plan` is an essential part of bug reports, where it is frequently described, and how it is provided in bug reports.

- RQ2: What is the best approach among discourse patterns, key-phrases, or keywords for identifying `Plans`?

This research question aims to determine, among these three possible strategies, which one is most effective at detecting `Plans` in sentences.

- RQ3: What is the minimal set of items for identifying `Plans`?

  This research question aims to understand, based on the identified strategy, what is a possible minimal list of items that will be good enough to identify `Plans` in bug reports.

1.1 Contribution of This Work

The contributions of this thesis are as follows:

- We came up with corpus of 126,308 bug reports derived from the MSR 2014 Challenge dataset, where each of the bug report comprises the bug id, bug title, bug description, bug comments, and the participants' names.
- We created a 1,149 manually annotated datasets of bug report sentences identifying which sentences contain `Plans.`
- We produced a fifty-one list of keywords and key-phrases that can be used to detect `Plans` bug reports.

1.2 Organization

The rest of this thesis proceeds as follows. In Chapter 2, we provide prior work on finding desired information from bug reports using different approaches. Chapter 3 presents our mythology of the proposed approach, where we will investigate different sections of bug reports to identify `Plans` and examine possible approaches to identify `Plans`. In Chapter 4, we present the outcomes of our investigation and discuss the performance of our approach. In Chapter 5, we present the implementation of the `Plan Labeller`. We conclude our study and outline future work in Chapter 6.

Chapter 2

Related Work

This chapter summarizes prior research based on bug report summarization including three techniques: patterns, key phrases, and keywords, and it discusses how useful these approaches could be in retrieving information from bug reports, classifying documents, and summarizing bug reports.

2.1 Bug Report Summarization

Rastkar et al. [30] proposed an automated bug report summarizing approach using supervised learning. Considering that bug reports are often written in a dialog fashion, they applied minutes and email threads summarizing algorithms to bug reports. They compiled handwritten summaries of thirty-six bug reports for assessment and training purposes. In their work, Jiang et al. [16] analyzed duplicated bug reports, discovered that there is a linguistic similarity between the duplicates and the original bug reports, and presented a bug report summary approach utilizing the PageRank algorithm using this relation.

In addition to supervised approaches, various unsupervised approaches were employed for summarizing. Mani et al. [26] suggested a method for reducing noise from bug reports, and using four types of unsupervised learning-based algorithms to analyze bug reports. Lotufo et al. [24] also suggested an unsupervised summarizing approach that made use of language similarities in bug reports by employing the PageRank algorithm. Li et al. [21] presented a deep learning-based strategy for summarization. Similarly, Liu et al. [22] also suggested a deep-learning-based technique for sentence summarization that assesses the

believability score.

2.1.1 Tagged Based Approaches

In the past, few researchers have also used tags or labels for bug report summarization. For instance, Boris et al. [4] proposed PorchLight, a tag-based interface using customizable query expressions that enables triagers to investigate, work with, and assign bugs in bug report groups. In their work, tags were primarily employed to label bug reports with common features and to create an effective interface for bug report triage. Song et al. [33] presented the Bee (Bug Report Analyzer) tool to help developers by organizing bug reports through the automatic detection and classification of terms to describe observed behavior, expected behavior, and actions to reproduce. In their study, they relied mostly on labels and comments in the bug report to notify reporters of missing components. Huai et al. [14] developed seven intention categories (Bug Description, Fix Solution, Opinion Expressed, Information Seeking, Information Giving, Meta/Code, and Emotion Expressed) and examined the relationship between bug report summaries and sentence intents. They primarily employed these intents to categorize the sentences of a bug report based on their intentions. Besides that, in many bug tracking systems, like the Google Chrome bug repository, bugs are labeled with the seven different categories of security, crash, regression, performance, usability, polish, and cleanup.

2.2 Information Extraction From Bug Reports

We present three strategies including patterns, key-phrases, and keywords, and discuss how these approaches were used in extracting information from bug reports by previous researchers.

2.2.1 Pattern Based Approaches

Patterns are linguistic rules that encapsulate sentences' syntax and semantics [5]. Patterns generate a set of rules that can be used to retrieve information from documents and

categorize sentences according to the rules. Patterns are utilized in several applications, most notably document classification, information retrieval, and neural networks.

Chaparro et al. [5] conducted research in which they detected information from bug reports using patterns. For their study, they concentrated mostly on bug descriptions and combed through them to identify observed behavior, expected behavior, and steps to reproduce. In order to do this, they manually analyzed reports from several bug tracking systems and developed 154 discourse patterns that can uniquely identify observed behavior, expected behavior, and steps to reproduce from bug report descriptions. Their research has shown that these patterns are extremely beneficial for identifying important information in bug reports. Even if developers submit bug reports in a free-form text format, it is still feasible to identify relevant information using specific patterns.

Sun et al. [35] proposed in their research an IR-based approach for bug localization that extracts bug patterns from version-related bug reports. Based on the information retrieval technique, they utilize version-related defect patterns to localize version-related defects. Then they retrieved the defect patterns of comparable bugs and ranked suspicious code based on the defect patterns and source code. Their empirical findings demonstrate that their strategy outperforms previous IR-based strategies.

In their work, Zhao et al. [37] proposed a hybrid classification method that employs linguistic patterns and machine/deep learning approaches to detect performance issue reports automatically. They created a comprehensive set of 80 heuristic linguistic patterns from developer-tagged performance concerns on the Apache Jira Platform. Then, based on these linguistic patterns, sentence-level and issue-level learning features were developed for training effective machine/deep learning classifiers.

Chawla et al. [6] proposed an automated approach that is capable of automatically classifying bug reports as functional bugs, security bugs, refactoring enhancements, etc. In their work, they used term frequency-inverse document frequency (TF-IDF) and latent semantic information (LSI) techniques for automatic bug labeling. Their research reveals that using

semantically related words acquired through LSI in combination with terms retrieved using TF-IDF improves the outcome of automatic labeling.

Even though these pattern-based approaches were very useful in classifying documents and retrieving information from bug reports, they still had some downsides. This pattern-based technique necessitates the identification of a set of rules and agreement on the patterns by a number of coders or researchers. Besides that, researchers have to spend a lot of time validating the patterns. Therefore, these approaches are very time-consuming and require a lot of manual effort.

2.2.2 Key-phrase Based Approaches

Keyphrases are a group of words that encapsulate the paragraph's main points. The primary objective of keyphrase extraction is to select a phrase that represents the primary substance of a text [34]. The key phrase makes information simple to manage, classify, and retrieve [23]. Application areas of keyphrase extraction include natural language processing (NLP), document categorization [15], information retrieval [1], and document indexing [10].

In their research, Jindal et al. [17] proposed a technique for bug report summarizing that makes use of both keyword-based features and sentence-based features to help retrieve the pertinent information. Rapid Automatic Keyword Extraction (RAKE) and term frequency-inverse document frequency (TF-IDF) were the two methods that they used for the extraction of keyword-based features. In their method, RAKE was used to extract phrases from the text, while TF-IDF was utilized to extract unigram words. In RAKE, each word is given a score based on how often it appears in a sentence. To compute the score of each keyword or keyword phrase, they computed the sum based on the content words in a text. Later, they utilized the extracted keyword and sentence features to develop rules, and based on the rules, they selected sentences to create an extractive summary.

He et al. [13] developed a deep learning-based method for determining and explaining

the validity of bug reports using only textual data. In their methodology, Convolutional neural network (CNN) was utilized to capture the contextual and semantic characteristics of bug reports. By backtracking CNN, they extracted valid bug report key phrases. Then, they manually analyzed key phrases and summarized valid bug report patterns from three aspects: Attachment, Environment, and Reproduce categories. Finally, based on these patterns and some related statistics, they provided reporters with suggestions on describing valid bug reports.

Roy and Rossi [18] suggested a strategy for feature selection that improves the predictive accuracy of severity prediction models. The development of features based on textual components and bi-grams, as well as the training of the Naive Bayes (NB) classifier with Mozilla and Eclipse data, demonstrated that the feature selection strategy improved the accuracy of predictions.

The disadvantages of key-phrase based techniques include the fact that, even when a key-phrase extraction technique is used, it still requires a lot of effort and necessitates calculating the frequency of phrases in order to identify them.

2.2.3 Keyword Based Approaches

Keywords are a subset of a document's words that describe the document's meaning [36]. Since the keyword is the smallest unit that can express the meaning of a document, it can be utilized by a variety of text mining applications, such as automatic indexing, automatic summarization, automatic classification, automatic clustering, automatic filtering, topic identification, and tracking. Therefore, keyword extraction can be regarded as the foundational technique for all document processing automation.

In their work on the automatic categorization of software bug reports, Otoom et al. [28] proposed a distinctive feature set based on the occurrences of particular keywords. They identified 15 keywords related to non-bug issues such as enhancement, improvement, and refactoring as the feature set. Using this feature set, they accurately classified newly

submitted bug reports into corrective or perfective classifications.

Peters et al. [29] presented Filtering And Ranking SECurity (FARSEC), a method for reducing the mislabeling of security bug reports using text-based prediction models. Their strategy is based on the fact that mislabeling is caused by the presence of security-related keywords in both security and non-security bug reports. Based on this discovery, they created a mechanism for automatically identifying keywords and ranking bug reports based on their likelihood of being classified as security bug reports (SBRs). Last but not least, they utilized FARSEC to decrease the presence of security-related terms. Prior to constructing prediction models, their methodology finds and eliminates non-security bug reports containing security-related keywords. With their approach, the class imbalance problem and the amount of mislabeled security flaws were drastically decreased.

Ekanayake et al. [9] in their research focused on the development of classification models for bug severity in unbalanced learning settings based on keywords. For their approach, they focused on the developer description of bug reports. Using the Rapid Keyword Extraction (RAKE) algorithm, they extracted keywords from the developer descriptions of bug reports. Then, the keywords were converted into numerical attributes, and combined with severity levels to construct datasets. They used these datasets and attributes to train classification models. The prediction skill of the models was evaluated using Area Under Recursive Operative Characteristics Curves (AUC) as the models were exposed to more skewed environments.

Badashian et al. [32] proposed a Bug Assignment (BA) approach in which their method constructs the expertise profile of project developers based on the textual elements of the bugs they have fixed in the past. Unlike traditional methods, however, their method considers only the programming keywords in these bug descriptions, with Stack Overflow serving as the vocabulary for these keywords. According to their technique, current expertise is more important than past expertise, hence their method measures the relevance of a developer's skill depending on how recently they have addressed a bug with keywords com-

parable to the bug at hand. They used Stack Overflow as a vocabulary of technical terms to determine the significance and specificity of keywords, and they combined them with the recency of developers' work in a scoring mechanism for BA. Their approach based on the textual elements shows that their model notably enhances the assignee recommendation accuracy.

Previous studies have shown the viability of using keyword-based techniques for a variety of text-mining tasks, including automated categorization and summarization. However, finding the ideal selection of keywords for a particular research activity is challenging. It still takes manual labor to examine and locate the proper collection of keywords, even when an algorithm is employed to extract keywords. Therefore, it is an extremely time- and labor-intensive strategy.

2.3 Summary

This chapter presented prior work on bug report summariztion and prior use of three possible strategies, specifically patterns, key-phrases, and keywords, which we investigate for our approach to identifying `Plans` in bug reports.

Chapter 3

Creating A Plan Labeller

We begin this chapter with defining what a `Plan` is and an overview of how we will identify a technique for detecting `Plans` in bug reports. Then we present the details of our approach.

3.1 Plan Labeller

A `Plan` is a comprehensive proposal for accomplishing something or a strategy for achieving a target [7]. In a software development context, a `Plan` is when developers create a solution for a problem, or propose a solution concept or recommendation. Therefore, in the context of our approach, identifying `Plan` as a `Solution` or `Suggestion` will make it conceptually simpler to identify `Plan` from sentences, as shown in figure 3.1. When a developer or user lacks sufficient confidence in their proposed idea, we can recognize a statement as a `Suggestion`. For example, in figure 3.1, "Try changing the parameter, it might fix your issue". Looking at "might fix", we can conclude that there is a possibility that the problem won't be resolved. On the other hand, for a `Solution`, the developer is sufficiently certain that their idea will work. For example, in figure 3.1, "Just change the parameter, it is working." Looking at "Just change", we may conclude that it will resolve the problem.

Our approach for identifying `Plans` from bug reports involves identifying a technique for labelling sentences that contain a `Plans`. We will initially examine bug reports from a data collection and identify sentences with `Plans`. From these sentences, a set of items will be found, either discourse patterns, key-phrases or keywords. The list of items will then

be validated and refined by iteratively applying the identified technique to other bug report subsets from the dataset.

We divide the proposed approach into two sections: the first discusses the approach for finding `Plans`, and the second discusses `Plan` identification refinement process. The first section utilizes the initial dataset, which helps to identify the initial list of items. For the second section, this initial list of items will be applied to the new dataset for validation, and during each iteration, the list of items will be updated or modified based on the performance of each iteration on the new dataset. Figure 3.2 shows the steps of our proposed approach.

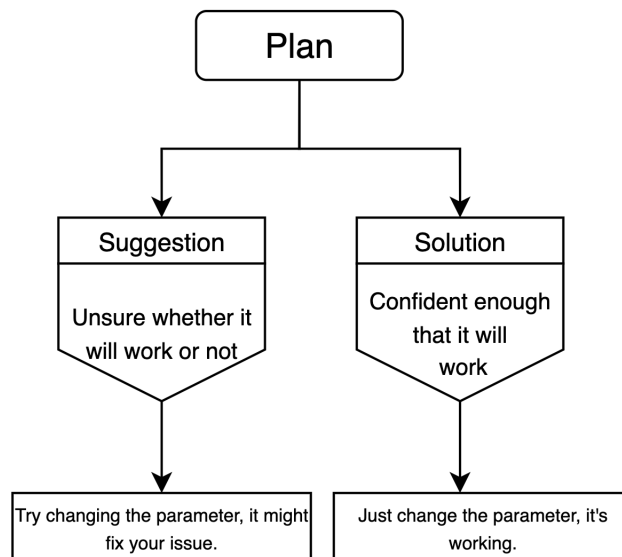

Figure 3.1: Defining Plans

3.2 Plan Detection Process

First, in order to identify `Plans` from bug reports, we discuss the various aspects of the bug reports that will be analyzed to find `Plans`. Then, we will discuss techniques for identifying `Plans` from sentences and the possible ways of categorizing the items used to detect `Plans`. Figure 3.3 demonstrates the procedure of identifying `Plans`.

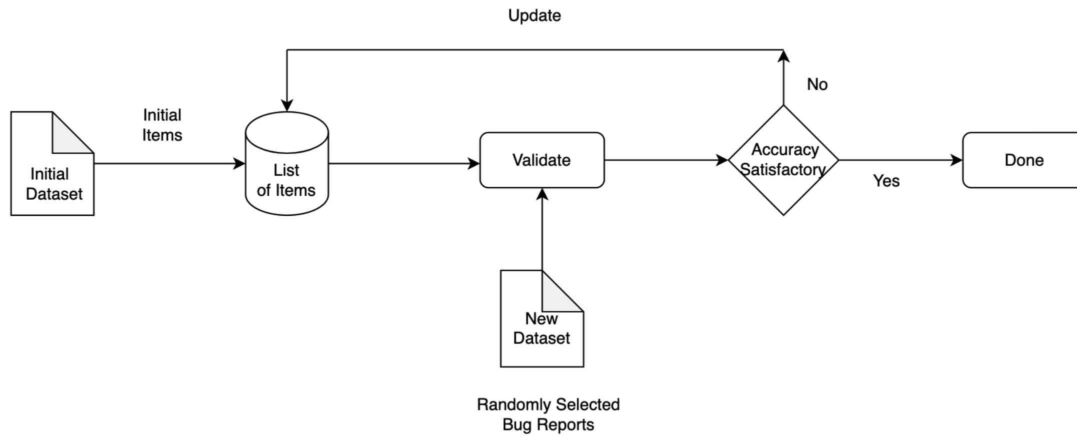

Figure 3.2: Proposed Approach.

3.2.1 Location of Plans in Bug Reports

First, we seek to understand what information bug reports contain, and which portion of the reports developers utilize to provide `Plans`.

A bug report facilitates communication between users and software developers regarding software flaws or feature requests. It includes a wide range of information, such as the report's identification number, its current state, the report's title, its description, and the conversations between users and developers to determine potential solutions or directions for resolving the issue. Typically, the report's title, description, and conversations between users and developers contain all the descriptive information regarding the bug report. Therefore, we will examine the Bug Title, Bug Description, and Bug Comments sections of each bug report to determine the existence of `Plans`. This investigation will help to find the answer of our RQ1.

Bug Report Title

The title contains a short summary of the problem, giving the reader a general idea of the bug. The title's information is presented in such a manner that the user or developer does not have to read the complete report to understand the problem. The title is just a noun phrase or words referring to the reports' topics [20].

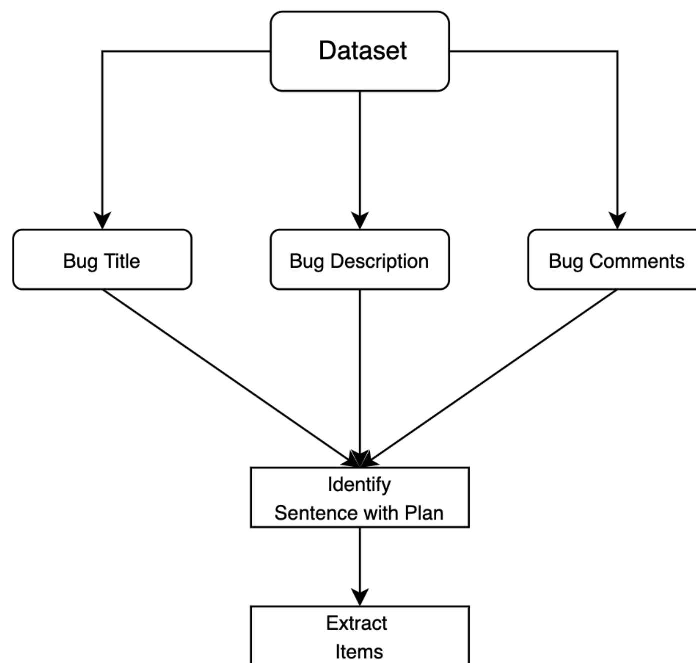

Figure 3.3: Plan detection Process

Bug Report Description

The description provides concise information about the defect. This information may include possible software faults or problems, as well as requests for software features or upgrades. If the reported issue is a bug or defect, the developers offer three crucial pieces of information: the actual outcome, intended behavior, and reproducibility methods. Alternately, if the issue pertains to feature requests or additions, they propose actions to enhance the software or advice on how to implement the requested feature.

Bug Report Comments

When developers have an opinion or information to share, they post a comment. For example, developers provide comments when the description's information is unclear to them or missing some information. Even if they can identify the issue, developers may provide a solution or instruction to solve it. These comments also include stack traces, links to other sources, and even off-topic phrases such as "Thanks for the help", and "Hello, I

hope you're doing well."

3.2.2 Detecting Sentences with Plan

After analyzing the corpus to determine which sections of bug reports include `Plans`, we move to finding sentences containing `Plans`. We investigate three approaches for locating `Plans` in bug report sentences which will help to answer our RQ2 regarding finding the possible approach for identifying `Plans`.

The first of these approaches is examining sentences and searching for a set of rules to discover `Plans`. Chaparro et al.'s [5] research demonstrated that developers follow well-defined standards in their descriptions to explain observed behavior, expected behavior, and steps to reproduce in the descriptions. Therefore, using a set of rules or patterns, it is possible to find important information in bug reports.

The second approach is examining the sentences to determine the use of key-phrases meaning a sequence of one or more words occurring together to identify `Plans`. In their method of sentence-based text summarization, Jindal et al. [17] demonstrated the use of both keywords and key-phrases to extract sentences from software defect reports. Their work shows that it is possible to use key-phrases to extract essential information from bug reports.

The last approach is examining the sentences and identifying a list of keywords, meaning a number of specific words, used for expressing `Plans` in sentences. While searching for a keyword in a sentence, the root form of the keyword will be identified and extracted, given that keywords can have different forms.

Regardless of which approaches will help to identify `Plans` from bug reports, stack traces, codes, and comments with quoted sentences will be ignored for the approach.

3.2.3 Categorization

Regardless of which of the three techniques for identifying `Plans` is used, the patterns, key-phrases, or keywords will be categorized. We describe our categorization approach in

the context of the use of keywords.

If a keyword is from the same bug report section as other keywords, they will be grouped together. A distinct list of keywords will be generated depending on the section of the bug report from which they were extracted. For instance, if the keyword is from the "Bug Report Comments" section, it will be grouped with the other keywords from the same location.

The keywords will next be grouped based on whether they appear in a single sentence or in multiple sentences. The number of sentences used will reflect the complexity of the `Plan`, with simple `Plans` appearing in a single sentence and complex `Plans` appearing in multiple sentences.

Lastly, keywords will be categorized based on their meaning and the manner in which developers use them to describe their `Plans`. For instance, if the keywords are used to describe similar intentions, they will be categorized together.

3.3 Plan Identification Refinement

To refine our `Plan` identification approach and to find the answer to RQ3 regarding which set of items will be minimal for identifying `Plans`, we randomly select reports from a dataset such that we have a 95% confidence level. Then, we apply the results of the previous iteration and examine them. We keep repeating the process until the improvement between iterations is minimal, or until a certain number of iterations have been completed.

During each iteration, the `Plan` identification approach will be refined based on four criteria: first, if the approach results in patterns, key-phrases, or keywords that are too generic or common. These would result in detecting too many sentences as `Plan`, leading to many false positives. Second, if some of them are not contributing frequently in detecting `Plan`, they will be considered too specialized. Third, if multiple items apply to the same sentence, we will remove one of them or try to combine them. Finally, if the collection of items is missing a lot of sentences with `Plan`, we will identify and introduce new items to our list for the next iteration.

3.4 Summary

In this chapter, we defined the meaning of `Plans` and introduced our approach for detecting `Plans` from bug reports. Then, we focused on the `Plan` detection process, which consists of a number of steps. The first step is examining different aspects of the bug reports to determine which section contains `Plans`. Then, we presented three possible techniques to detect sentences with `Plans`. Lastly, we described the ways of categorizing the items to find `Plans`. At the end, we focused on refining the `Plan` identification process, where we discussed our iteration process and after each iteration, how we would refine the items.

Chapter 4

Results and Evaluation

In this chapter, first, we discuss the data sources we used for our investigation. Then, we present the results of our investigation of which parts of a bug report contain `Plans`. Next, we present the results from our investigation of how to identify `Plans` in bug reports. Finally, we present the outcome of iteratively refining the `Plan` identification approach.

4.1 Data source

For our approach, we used bug reports from two different data sources, such as: Rastkar et al.'s [30, 31] dataset and the Mining Software Repositories (MSR) dataset [12]. For our approach, we first wanted to investigate whether it was possible to find `Plans` in a bug report or not. Since the approach of identifying `Plans` from sentences has never been done before, we wanted to start with a smaller dataset that contains bug reports from a wide variety of projects so that we could analyze them within a short period of time and identify lists of items with a lot of variation. Therefore, to identify `Plans` location in bug reports and to find out which of the three approaches, whether patterns, key-phrase or keywords, can be used to extract `Plans`, we use Rastkar et al.'s [30, 31] dataset. For conducting iterations and refining the `Plan` identification process, we use the MSR dataset. The MSR 2014 Challenge data collection was selected to verify the accuracy of the items used for `Plan` identification over a wide range of bug reports.

4.1.1 Rastkar's Dataset

We chose to use the bug report corpus curated by Rastkar et al.'s [30, 31] to investigate where developers provide `Plans` and how to identify them. The choice of using this corpus was made for two reasons. To begin with, the dataset contains bug reports from a broad variety of different projects. It was essential for our approach to make use of a dataset of this kind in order to determine a set of keywords that are capable of generating precise results for bug reports taken from a wide range of sources. Second, the dataset was previously utilized by Galappaththi et al. [11] and other researchers [25, 26] for a bug summary approach, meaning that the dataset's content has already been validated.

The corpus consists of thirty-six (36) bug reports with a total 2,361 sentences collected from four distinct open-source projects (nine reports from each project), including Mozilla, Eclipse, KDE, and Gnome. All the bug reports are stored in XML format. Each of the reports contains information such as the `Bug Id` to indicate which particular bug report is currently being accessed, `Title`, which provides short detail regarding the issue, and all the `Comments` to show the conversation between the developers.

Figure 4.1 shows the schema of Rastkar et al.'s [30, 31] corpus taken from Galappaththi et al's [11] work. Each bug report's comments are presented in a format resembling a conversation between two or more individuals who take turns talking [11]. Therefore, each comment is considered a turn, and each turn includes the contributor's name, the time the comment was made, and the text of the individual sentences from the entire comment. One thing to note, the first comment is always the description of the bug reports. No distinct attributes were used to differentiate descriptions from the comments. Figure 4.2 represents a sample from the Rastkar et al.'s [30, 31] dataset.

4.1.2 The MSR Dataset

To assess the effectiveness of our approach, the prime requirement is a corpus of bug reports. For this, a corpus of bug reports from the Mining Software Repositories (MSR)

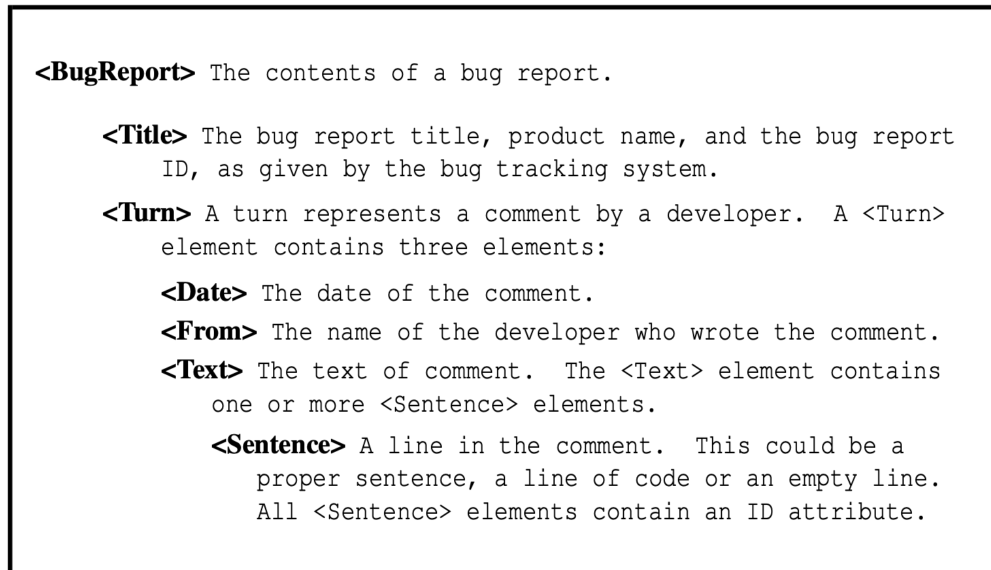

Figure 4.1: General format of the bug reports captured from Galappaththi et al's [11] work.

2014 Challenge [12] was selected. The dataset includes ninety projects and their forks for the most popular programming languages on GitHub. The corpus consist of 126,308 bug reports with 583,794 comments (6,572,498 sentences). The data for each project includes problems, pull requests, organizations, followers, and labels.

The dataset from the MSR 2014 Challenge [12] is available in two distinct formats: a MongoDB database dump and a MySQL database dump. We chose to utilize the MongoDB database dump.

Data Processing and Extraction

The MongoDB database dump contains sixteen distinct data tables, including commit comments, issue comments, repo labels, and many others. Each table contains a wide variety of information from different parts of the bug reports. However, only the "issues" and "issue_comments" data tables include the information required for our investigation. The "issues" table contains a summary statement of the bug and the titles, and the "issue_comments" table contains the discussion in the form of comments made by the participants.

```
</BugReport>
<BugReport ID = "3">
    <Title>"(491925) Firefox - Disable multitouch \"rotate\" gesture for cycling tabs"</Title>
    <Turn>
        <Date>'2009-05-07 13:09:47'</Date>
        <From>'Justin Dolske'</From>
        <Text>
            <Sentence ID = "1.1"> I've noticed that I frequently trigger the rotate gesture
            accidentally, usually while scrolling.</Sentence>
            <Sentence ID = "1.2"> Gestures have improved since the original landing (when
            triggering the wrong gesture was really easy), but this articular gesture is still
            problematic.</Sentence>
            <Sentence ID = "1.3"> The basic reasoning is that it's highly disruptive to be switched
            to another tab when you're not expecting it.</Sentence>
            <Sentence ID = "1.4"> When it happens, you don't know what just happened until you
            notice that you're on some entirely different page, that's randomly to the left or
            right (1 or more tabs) from the page you thought you were on.</Sentence>
            <Sentence ID = "1.5"> I'm don't think the rotate gesture for switching tabs is nearly
            as useful, discoverable, or a good fit as the other gestures are.</Sentence>
            <Sentence ID = "1.6"> So, given this problem, we should just disable it for 3.5. </
            Sentence>
            <Sentence ID = "1.7"> [I'd also be open to tweaking it to make it much harder to
            trigger accidentally, dunno if that's possible.]</Sentence>
        </Text>
    </Turn>
    <Turn>
        <Date>'2009-05-07 13:20:26'</Date>
        <From>'Henrik Skupin'</From>
        <Text>
            <Sentence ID = "2.1"> See also bug 461376. </Sentence>
            <Sentence ID = "2.2"> Just play around with browser.gesture.twist.* in about:config as
            a temporary workaround.</Sentence>
        </Text>
    </Turn>
```


Figure 4.2: A sample from the Rastkar et al.'s [30, 31] dataset

To access those tables from the MongoDB database, we use Pymongo [27]. From both of the tables we extracted the following information:

- The bug “Id”, a bug identification number.
- The bug “_Id”, a unique identification number generated by MongoDB.
- The bug “Title,” a summary of the issue.
- The “User,” name of the participants who contributed in the bug report.
- The “body,” a summary statement of the bug and the description or the comments made by the developers.

At the end, we combined both tables, along with the properties that were extracted, and saved the results in a new table. From there, with the help of NLTK [3], we split all the text of the bug description and comments into sentences. Developers sometimes provide `Plans` using multiple sentences, especially when the problem is complex. If it is kept in paragraph format, annotating them with the `Plan Labeller` will be a difficult task to achieve.

We extracted 126,308 bug reports in JSON format, where each bug report has a “title,” “participant’s name,” “description,” and “comments” for that particular bug report.

4.2 Bug Report Sections With Plans

To understand which portion of the reports developers utilize to provide `Plans`, we manually analyzed Bug Reports Title, Bug Reports Description, and Bug Reports Comments from Rastkar’s dataset.

Bug Reports Title

As the intent of the title is to provide a short description of the issue so that the reader does not have to read the entire report to understand the issue, after examining the bug report titles, we found that the `Plan` does not appear there.

Bug Report Description

After analyzing the bug descriptions, we found that bug descriptions sometimes contain `Plans`, especially if the bug report is regarding feature requests or enhancements. In addition to the actual outcome, expected behavior, or steps to reproduce in bug description defect-related reports, developers even include their ideas or ways to solve the issue. Figure 4.3 shows an example of a bug report description that contains `Plans`.

Bug Report Comments

After going through the comments of the bug reports, we found that comments do contain `Plan`, but the way they are provided varies. Figure 4.4 shows an example of a bug

```xml
</BugReport>
<BugReport ID = "5">
    <Title>"(224588) Eclipse - [Patch] Provide an information how many lines does the patch
    change"</Title>
    <Turn>
        <Date>'2008-03-28 09:19:00'</Date>
        <From>'Tomasz Zarna'</From>
        <Text>
            <Sentence ID = "1.1"> Inspired by Martin Oberhuber's mail about his \"lsc\" script for
            counting lines in a patch[1], I though that it maybe be worthwhile to embed such thing
            in the Apply Patch wizard itself. </Sentence>
            <Sentence ID = "1.2"> imo using regexp should be enough here, especially due to the
            fact that we don't want to make any additional dependencies. </Sentence>
            <Sentence ID = "1.3"> So, my proposal is to allow to specify a regexp rule for lines
            that should be count as a \"real contribution\".</Sentence>
            <Sentence ID = "1.4"> This is what first came to my head:</Sentence>
            <Sentence ID = "1.5"> /-\\\\+? (\\s*\\S+\\s*)+$</Sentence>
            <Sentence ID = "1.6"> which means \"Count only lines starting with single '+' and a
            space. </Sentence>
            <Sentence ID = "1.7"> The line must also have at least one non-whitespace character\".
            </Sentence>
            <Sentence ID = "1.8"> I think this is more/less what Martin's script does.</Sentence>
            <Sentence ID = "1.9"> All lines that match the above pattern would be sum up and the
            info would be displayed somewhere on the Apply Patch wizard. </Sentence>
            <Sentence ID = "1.10"> How does it sound? Martin?</Sentence>
            <Sentence ID = "1.11"> Again, I think it's a brilliant idea Martin.</Sentence>
            <Sentence ID = "1.12"> [1] sent to eclipse.org-committers</Sentence>
        </Text>
    </Turn>
```

Figure 4.3: An example of `Plan` in bug description.

report comment which contains `Plans`.

After examining the different sections of bug reports, we found that Bug Description and Bug Comments contains `Plan` which provides an answer to our RQ1.

4.3 Plan Identification

In order to detect `Plans` in bug reports sentences, we investigated three possible approaches beginning with the most general and narrowing to the most specific. These approaches are patterns, key-phrases and keywords.

4.3.1 Patterns

We began our investigation by examining discourse pattern-based approach for extracting `Plans` from sentences, similar to that used by Chaparro et al.'s [5]. As developers utilize a free-form text format, we were unable to find a set of rules or patterns that uniquely identify `Plans`.

```
</BugReport>
<BugReport ID = "27">
    <Title>"(188311) KDE - The applet panel should not overlap applets "</Title>
    <Turn>
        <Date>'2009-03-28 11:35:10'</Date>
        <From>'mangus'</From>
        <Text>
            <Sentence ID = "1.1"> version:           svn (using Devel)</Sentence>
            <Sentence ID = "1.2"> OS:                Linux</Sentence>
            <Sentence ID = "1.3"> Installed from:    Compiled sources</Sentence>
            <Sentence ID = "1.4"> In amarok2-svn I like the the new contextview , but I found the
            new bottom bar for managing applets annoying , as it covers parts of other applets
            sometimes , like lyrics one , so that you miss a part of it. </Sentence>
            <Sentence ID = "1.5"> Could be handy to have it appear and desappear onmouseover.</
            Sentence>
            <Sentence ID = "1.6"> thanks</Sentence>
        </Text>
    </Turn>
    <Turn>
        <Date>'2009-03-28 14:53:55'</Date>
        <From>'Dan Meltzer'</From>
        <Text>
            <Sentence ID = "2.1"> The real solution is to make it not cover applets, not make it
            appear/disappear on mouse over.</Sentence>
        </Text>
    </Turn>
```

Figure 4.4: An example of `Plan` in comments.

4.3.2 Key-phrases

Next, we examined a key-phrase based method for identifying sentences containing a `Plan`. However, identifying key-phrases was challenging since developers use a different set of key-phrases depending on the issue. Each bug has a unique `Plans` and to express that, developers hardly use the same key-phrases. In spite of that, we were able to identify some key-phrases to detect `Plans`.

4.3.3 Keywords

Lastly, we examined the sentences from bug reports and found that developers use a specific list of keywords to express `Plans`. While extracting a keyword from a sentence, we extract the root form, given that keywords can have different forms. We have seen that sometimes developers do not express their `Plan` using a single sentence. Especially when the problem is complex and requires multiple sentences to express, they use multiple sentences, or even enumerated lists or paragraphs, to address it. They link those sentences using

Table 4.1: Commonly Used Keywords and Key-phrases

Keyword Name	Example
Work	If you add the below snippet it will **work** and solve the issue.
Fix	One possible **fix** is just change the parameter.
I Think	Rather than add multiple ways to do the same thing in the main repo **I think** it's best if this one remains a separate project.
Should Be	You **should be** overriding to_param in order to change the url.
However	Naming them doesn't resolve the issue **however**, 'environment = environment()' is the only way I can see it working at the moment.
Therefore	**Therefore** I propose following syntax for global functions: "'@covers ::function_name"'.

conjunctive adverbs such as `However`, `Therefore`, and `Since` to add multiple sentences. Therefore, we have also extracted some of the keywords, which will be able to detect both sentences rather than one sentence which does not contain the full `Plans`. Figure 4.5 shows an example of detecting keywords from the bug's comments.

4.3.4 Summary

Although we were able to identify some key-phrases, we found that they were insufficient to detect `Plans` in general. Therefore, we focused on a keyword-based strategy that incorporates both individual keywords and key-phrases. This result provides an answer to our RQ2.

We identified seventy-three unique items, including single keywords, key phrases, and a few conjunctive adverbs. Among the keywords, we found that `WORK`, `JUST` and `FIX` are the most commonly used keywords, `I THINK` and `SHOULD BE` are the most commonly used key-phrases, and `HOWEVER`, `THEREFORE` are the most commonly used conjunctive adverbs. Table 4.1 shows an example of the most used keywords and key-phrases from the Rastkar et al.'s [30, 31] dataset.

```
</BugReport>
<BugReport ID = "27">
    <Title>"(188311) KDE - The applet panel should not overlap applets "</Title>
    <Turn>
        <Date>'2009-03-28 11:35:10'</Date>
        <From>'mangus'</From>
        <Text>
            <Sentence ID = "1.1"> version:           svn (using Devel)</Sentence>
            <Sentence ID = "1.2"> OS:                Linux</Sentence>
            <Sentence ID = "1.3"> Installed from:    Compiled sources</Sentence>
            <Sentence ID = "1.4"> In amarok2-svn I like the the new contextview , but I found the
            new bottom bar for managing applets annoying , as it covers parts of other applets
            sometimes , like lyrics one , so that you miss a part of it. </Sentence>
            <Sentence ID = "1.5"> Could be handy to have it appear and desappear onmouseover.</
            Sentence>
            <Sentence ID = "1.6"> thanks</Sentence>
        </Text>
    </Turn>
    <Turn>
        <Date>'2009-03-28 14:53:55'</Date>
        <From>'Dan Meltzer'</From>
        <Text>
            <Sentence ID = "2.1"> The real solution is to make it not cover applets, not make it
            appear/disappear on mouse over.</Sentence>
        </Text>
    </Turn>
```

Figure 4.5: Detecting keywords from the bug comments

4.4 Refining The Plan Detection Approach

To test the accuracy of our identified keywords and key-phrases, we extracted 126,308 bug reports from the MSR dataset [12]. From there, for each iteration, we randomly select 383 bug reports (95% confidence level) and apply the identified keywords and key-phrases to these bug reports. Based on the outcome of the iteration, the keywords and key-phrases are analyzed and refined.

4.4.1 Evaluation Metrics

We assessed the effectiveness of a keyword and key-phrase set using three performance metrics: Precision, Recall, and F-score. For our approach, we defined True Positive (TP) as "Number of Sentences containing `Plans` correctly identified by the `Plan Labeller`", False Positive (FP) as "Number of Sentences not containing `Plans` but identified as `Plan` by the `Plan Labeller`" and False Negative (FN) as "Number of sentences containing `Plans` missed by the `Plan Labeller`." F1-score indicates the harmonic mean of precision and recall depicts the overall performance of the proposed approach.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$F1 - score = \frac{2(Precision * Recall)}{Precision + Recall}$$

4.4.2 First Iteration Results

For the first iteration, we used the seventy-three keywords and key-phrases that we identified from examining the Rastkar's dataset and applied them to a randomly selected 383 bug reports from the MSR dataset. To better comprehend the efficacy of keywords and key-phrases, we separated the iterations into two sections: one for the description and the other for the comments. Figure 4.6 shows the result of first iteration.

For 1st Iteration	Keywords Used	New Keywords Added	Keywords Removed	Title	TP	FP	FN	Total	Precision	Recall
	73	0	0	Description	430	401	26	1264	51.74%	94.29%
				Comments	1277	833	94	3429	60.52%	93.14%

Figure 4.6: The results of first Iteration

The description part from the 383 bug reports contained a total of 1,264 sentences. Our keyword and key-phrase set correctly identified 430 sentences (34%) as having `Plans`. The incorrectly identified sentences were 401 sentences 32%. This indicates that the keywords and key-phrases set misidentified a large number of sentences as `Plan` that do not include `Plans`. The number of sentences with `Plan` that were not identified was lower, which was only 26 (around 2%) out of the 1,264 sentences.

The comment part of all 383 bug reports had a total of 3,429 sentences. Using the keyword and key-phrase set, 1,277 sentences (37%) were correctly identified as a `Plan`. The incorrectly labeled sentences were 833 sentences (24%), and the number of missed

sentences was 94 (around 3%).

Discussion

Examining the precision and recall, we observed that the keyword and key-phrase set had a high recall for both descriptions and comments. However, precision was found to be in the 50-60% range. To determine the cause of the lower precision, we analyzed all of the false positives (sentences incorrectly identified as `Plan`) and determined that the majority of the keywords and key-phrases initially identified by Rastkar's dataset were far too generic, meaning they are also used to express other concepts in bug reports. Therefore, we removed twenty of the keywords and key-phrases that were incorrectly labeling sentences. We also studied the false negatives (sentences missed by the keywords and key-phrases) and discovered three additional keywords `Keep`, `Might`, and `Wonder`, which we feel will assist in identifying `Plans`.

4.4.3 Second Iteration Results

For the second iteration, we used fifty-six keywords and key-phrases after removing twenty keywords and adding three new keywords from the first iteration. This keyword set was applied to a new set of randomly selected 383 bug reports from the MSR dataset. Figure 4.7 shows the result of second iteration.

For 2nd Iteration	Keywords Used	New Keywords Added	Keywords Removed	Title	TP	FP	FN	Total	Precision	Recall
	56	3	20	**Description**	398	209	119	1199	65.56%	76.98%
				Comments	841	460	237	2931	64.64%	78.80%

Figure 4.7: The results of second Iteration

After using the revised keyword and key-phrase set, we observed that out of 1,199 sentences of bug descriptions, 398 sentences (33%) were correctly identified sentences from the first iteration and 209 sentences (17%) were not. If we compare it with the first iteration, the number of correctly identified sentences dropped from 34% to 33%, but the the number

of incorrectly identified sentences decreased from 32% to 17%. Even though the correctly identified sentences decreased by 1%, incorrectly identified sentences decreased to almost half which means our list of keywords and key-phrases detects `Plans` more accurately than the first iteration. However, the missed sentences have drastically increased to 119 (10%), whereas for the first iteration, it was just 26 sentences (2%).

The comment part of all 383 bug reports had a total of 2,931 sentences. From there, 841 sentences (29%) were correctly identified as `Plan`. Similar to the description part of our second iteration, the incorrectly identified sentences has decreased here as well, to 460 sentences (16%). Compare to the comment part of the first iteration, the correctly identified dropped from 37% to 29%, but most importantly, the incorrectly identified sentences dropped significantly from 24% to 16%. Even though, the number of missed sentences rose to 8% from 3%, overall the updated list of keywords and key-phrases also performed better here.

Discussion

After calculating the precision and recall, we saw that the precision had grown for both the description and the comments, where it was around 65% for the description and 64% for the comments. In contrast to the initial iteration, the recall for both descriptions and comments reduced dramatically. To determine the reason for the lower recall, we analyzed and found that it was due to the increased number of missed sentences containing `Plans`. To balance that, we went through the missed ones and added two new key-phrases, `Can you` or `You Can`, and `I think`. Among these two, `I Think` was omitted after the first iteration, but after analyzing the missed sentences, we discovered that it helps more in detecting `Plan` than the incorrect one. Therefore, we reintroduced this key-phrase in order to improve the recall. Even though the precision was increased, we still analyzed the incorrectly detected sentences and discovered seven keywords and key-phrases, such as `Make sure`, `Update`, `Close`, `Attempt`, `Bug`, `Check`, and `Show`, which were responsible for detecting incorrect

sentences. Therefore, we removed seven and added two, which we believe will help identify `Plans` more accurately.

4.4.4 Third Iteration Results

For the third iteration, we used fifty-one keywords and key-phrases after removing seven and adding two new to the list from the second iteration. These were applied to a newly selected random set of 383 bug reports. Figure 4.8 shows the result of the third iteration.

For 3rd Iteration	Keywords Used	New Keywords Added	Keywords Removed	Title	TP	FP	FN	Total	Precision	Recall
	51	2	7	Description	313	178	81	1205	63.74%	79.44%
				Comments	1017	476	206	3537	68.11%	83.15%

Figure 4.8: The results of third Iteration

After applying the new keywords and key-phrase, we observed that, out of 1,205 bug description sentences, 313 (26%) were successfully labelled as `Plan`, whereas 178 (15%) were incorrectly identified. This indicates that the ratio of incorrectly identified sentences has reduced by 2%, as well as the correctly identified by 7%. However, fewer `Plan` sentences were missed in the third iteration compared to the second as the incorrectly identified sentence rate dropped to 7%. This means that the updated set of keywords and key-phrases can find more of the sentences that were missed in the second iteration. However, this comes at a cost of a lower correctly identified sentence rate.

The comment portion from the selected 383 reports has a total of 3,537 sentences. From there 1017 (29%) sentences were correctly identified as `Plan` and 476 (13%) sentences were not. Compare to the second iterations, comment part, the number of correctly identified sentences is similar in both iterations. However, the incorrectly identified sentences dropped 3%. Even the number of missed sentence dropped, from 8% to 6%. This indicates that our updated set of keywords and key-phrases worked even better then iteration two.

Discussion

After assessing the precision and recall, we found that the precision for comments has grown to 68%, while the precision for descriptions has declined to 63%. However, the recall has risen for both the description and the comments compared to the second iteration, although these are still lower than the first iteration. From the third iteration, we can see that the list of keywords and key-phrase is performing well, as total precision is the best of the three iterations, and the total recall is rising.

4.4.5 Discussion

Table 4.2 shows the precision, recall, and F1 score of each iteration. After three iterations, our keyword or key-phrase set has an F1 score of 73%, which means they were able to detect `Plan` relatively effectively.

Looking at Iteration one, we see that the precision was the lowest of the iterations, and the recall was the highest. The main reason for these results is that the keywords or key-phrases we used, which were identified from Rastkar's dataset were too general. The Rastkar's dataset was comparatively smaller then those of the MSR dataset, so the data would not represent as many variations in how `Plans` are expressed.

If we compare the results between iteration two and iteration three, overall the precision, recall and F1-score rose in a stable way. Based on the performance of iteration three, we believe we have identified the minimal list of keywords and key-phrases which can identify `Plans` from sentences. Therefore, we did not continue further iterations.

During our iterations, we identified three keywords that appeared in both sentences that were and were not `Plans`. In the second iteration, we wanted to remove these keywords but found that doing so will have a significant negative effect. The performance improvements for Iteration three show that these keywords are more helpful than harmful. For example, we can see that in the sentence "Unfortunately, this will not `work` in the current version of Requests, which insists you have both a netloc (e.g., hostname) and a path." from Bug

Table 4.2: Precision, Recall and F1 Score from the iterations

Iterations	Title	Precision	Recall	F1 Score
1st Iteration	Description	51.74%	94.29%	66.82%
	Comments	60.52%	93.14%	73.34%
	Total	58.04%	93.43%	71.60%
2nd Iteration	Description	65.56%	76.98%	70.81%
	Comments	64.64%	78.80%	71.02%
	Total	64.93%	77.68%	70.74%
3rd Iteration	Description	63.74%	79.44%	70.72%
	Comments	68.11%	83.15%	74.88%
	Total	67.03%	82.25%	73.86%

#6762208, the word "Work" helps identify a `Plan`, but in the sentence "`Works` all right on Nexus 4 running 4.3." from Bug #13627177, it does not.

4.5 Keywords And Key-phrases Categorization

Having determined the set of keywords and key-phrases for detecting `Plans`, we started categorize them.

Initially, we analyzed the keywords and key-phrases to determine if they came from the same source so that we could categorize them accordingly. We observed that developers use the same set of keywords and key-phrases to describe `Plans` in both descriptions and comments; they do not use a different set of keywords and key-phrases in different parts of the reports. Therefore, we were unable to differentiate keywords and key-phrases depending on their source and group them for use in a particular part of a bug report.

Then, we divided the keywords and key-phrases into two categories based on how they appear in the sentences. The first category is `Sentence Level`, and the other one is `Paragraph Level`. `Sentence Level` keywords and key-phrases express a `Solution` or `Suggestion` in a single sentence, whereas `Paragraph Level` keywords and key-phrases require multiple sentences to express it. Most of the keywords and key-phrases are part of the `Sentence Level` category (91%), and a few are part of the `Paragraph Level`,

Table 4.3: Keyword And Key-phrase Categories

Category Name	Reason For Naming
Alternative	When developers have more than one plan.
Clarification	When developers have additional information to provide for a better understanding plan.
Command	When the developer is confident enough about the plan.
Conclusion	When developers have a plan, but it might cause some issues with the software later.
Desire	When the developer has an idea for the plan but does not know how to do it.
Judgement	When developers provide a plan based on judging others' plans.
Noise	When developers use a common word to express both plans and normal statements.
Phrase	When developers use more than one word, that occurs together to express a plan
Reference	Whenever developers refer to documents or links for Plans
Solution	When the developer has the exact plan and is confident enough that it will work.
Suggestion	When the developer has an idea for a plan but is unsure whether it will work,

category (9%). Also, we identified some keywords and key-phrases that are part of both categories (8%).

Lastly, we applied a card sorting technique [8] to categorize keywords and key-phrases into different groups based on their meaning and how developers use them to describe their `Plans`. Five individuals participated in the card sorting [8] and arrived at eleven categories. For example, the `Reference` category contains six keywords `Duplicate`, `Patch`, `Attach`, `Include`, `Read`, `Document` and these keywords mainly appear whenever someone refers to something like documents or links. Similarly, another category, called `Noise`, contains keywords that appear in both sentences, which includes `Plan`, and those that do not. Table 4.3 shows the names of all eleven categories and the reason for their naming.

4.6 Final List of Keywords and Key-phrases

During each iteration, a different set of keywords and key-phrases was used to test the performance of our `Plan` identification process. Figure 4.9 shows the number of keywords utilized in each iteration. After examining all iterations, fifty-one keywords were found that effectively identify `Plans` in a sentence, which answers our RQ3. Table 4.4 shows the fifty-one keywords and key-phrases with their category. A complete list of all keywords and key-phrases used in our investigation for each iteration can be found in Appendix A.

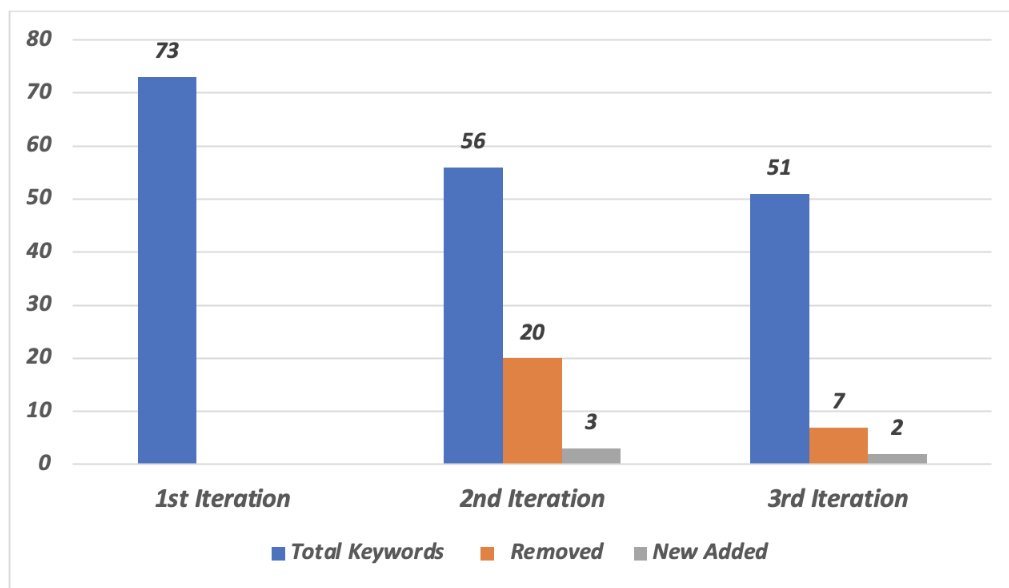

Figure 4.9: Number of Keywords And Key-phrases Used Each Iteration

4.7 Threats to Validity

In this work, we evaluated the effectiveness of our approach based on our manual labelling. Due to time constraints, we were not able to perform an external validation of our iteration result. Conducting such a validation is a next step. It will help to reduce any errors made during manual labeling and will possibly generate a more accurate annotated dataset.

During each iteration, numerous sentences were detected as a `Plan` by our keywords and key-phrases, but in some cases it was difficult for us to decide whether they were actually

Table 4.4: Final List of Keywords And Key-phrases With Category

Category Name	Keywords and Key-phrases
Alternative	However, Instead, Other, Although, Whether, Otherwise
Clarification	In Addition, For Example, Option, Because, Actual
Command	Improve, Make, Found, Implement, Create, Approach, Change, Keep, Select
Conclusion	In fact, Since, Therefore
Desire	Want, Need, Miss, Support
Judgement	Possible, Reasonable, Probably, Seem, Easy, Wonder, Agree, Might, Far
Noise	Work, Just, Fix
Phrase	I Think, Can-Could You/ You Can-Could
Reference	Patch, Attach, Include, Read, Document, Duplicate
Solution	Solve, Solution, Resolve
Suggestion	Suggest

part of a `Plan` or not. We defined a `Plan` as a `Suggestion` or `Solution`. However, after going through the bug reports, the question arises of whether we should always count a `Suggestion` or `Solution` as a `Plan`. We found cases where a `Suggestion` or `Solution` is not directly related to the particular bug report, and it could be difficult to decide whether we should consider the sentence as part of a `Plan`. For example from Bug #6126149, we found the sentence "Do not report issues on $github \backslash nPlease$ use the XBMC $forum \backslash n \backslash nPlease$ close." and in Bug #2573248, the sentence "This can be closed since it was fixed." These types of sentences are `Suggestion` or `Solution` but they are not `Plans` about the particular bug report.

4.8 Summary

In this chapter, we began with presenting the data sources we used for our approach. We used Rastkar's dataset for our initial `Plan` identification process. From there, we identified the location of `Plans` in a bug report and found that developers use a certain list of keywords and key-phrases for expressing `Plans` in sentences. Then, we used the MSR 2014 Challenge dataset for our `Plan` identification refinement process. From there, for each itera-

tion, 383 bug reports were randomly selected and the list of keywords and key-phrases was improved. After three iterations, we arrived at fifty-one list of keywords and key-phrase and the results showed that this list is effective at identifying `Plan` in sentences. Lastly, we divide the keywords and key-phrases into eleven categories based on their meaning and how they appear in a sentence.

Chapter 5

Implementation of the Plan Labeller

This chapter describes our implementation of a `Plan Labeller` based on the results from chapter 4. Later, we present the results of the implementation.

5.1 An Automated Approach

Our manual approach of detecting `Plans` from bug reports revealed that it is possible to identify sentences with `Plans` using a set of keywords and key-phrases. Our study indicated that developers use a relatively limited set of keywords and key-phrases to express `Plans` in bug reports. The results from the manual approach motivates the need for an automatic approach to detect sentences with `Plans` in bug reports. Therefore, we developed and tested an automatic approach for implementing the `Plan Labeller`, using regular expressions.

The `Plan Labeller` uses regular expressions to detect if sentences from bug descriptions or bug comments contain a `Plan` or not. The regular expressions encode our final list of items, the keywords and key-phrases explicitly found to refer `Plans`. The label `Plan` is assigned to a sentence if it contains any of the fifty-one keywords or key-phrases, such as "approach," "patch," and "work". During the implementation, we did not perform stop words removal as some of the key-phrases include stop words.

While examining the bug reports, we identified various types of sentences that contain `Plan`. For example "This patch should fix GimpZoomPreview to handle layers with offsets with selections." (from bug report GIMP #156905). Therefore, we use the regular expression `.*patch.*` to identify such sentences. Similarly, we have generated regular

expressions for the other keywords and key-phrases which can uniquely identify a sentence with `Plans`.

5.1.1 Applying The Automated Approach

To test the accuracy of the automated approach, we randomly selected fifty bug reports from the MSR dataset that were not previously used in our work. For each bug report, the `Plan Labeller` goes through the sentence of the bug description and bug comments, and, using the regular expression, it will try to match one or more of the fifty-one keywords and key-phrases. If any of the sentences contain a keyword or key-phrase that matches with our regular expression, the `Plan Labeller` will label it as `plan:1` otherwise label it as `plan:0`. Figure 5.1 shows an example of a labeled bug report by the `Plan Labeller` from the MSR dataset.


```
{
    "id": 401279,
    "title": "Error building lilypond 2.13.36",
    "user": {
        "login": "cayblood"
    },
    "body": [
        {
            "sentence_id": 1.1,
            "sentence": "I have a configure error when I try to install lilypond.",
            "plan": 0
        },
        {
            "sentence_id": 1.2,
            "sentence": "It looks like it is missing some font related tools called metafont and metapost.",
            "plan": 1
        }
    ],
    "comment_list": [
        {
            "id": 518645,
            "user": {
                "login": "adamv"
            },
            "body": [
                {
                    "sentence_id": 2.1,
                    "sentence": "You need to install TeX first.",
                    "plan": 1
                }
```


Figure 5.1: A labeled bug report by the Plan Labeller

Automated Approach	Title	TP	FP	FN	Total	Precision	Recall
	Description	32	17	13	99	65.30%	71.11%
	Comments	69	38	18	287	64.48%	79.31%

Figure 5.2: The result of automated approach

5.1.2 Results of Applying The Automated Approach

Figure 5.2 shows the results of applying this automated approach. The description part of the 50 bug reports contained a total of 99 sentences, and our automated `Plan Labeller` correctly identified 32 sentences (32%) as `Plan` and 17 sentences (17%) were incorrectly identified as `Plan`. Since the number of incorrectly identified sentences is almost half as the correctly identified ones, this indicates that the `Plan Labeller` is performing well.

Similarly, the comment part of all 50 bug reports had a total of 287 sentences. Using the automated `Plan Labeller`, 69 sentences (24%) were correctly identified as a `Plan` and the incorrectly labeled sentences were 38 sentences (13%). Similar to the description part, these results indicate that the `Plan Labeller` also performed well here.

5.2 Discussion

Table 5.1 shows the precision, recall, and F1 score of both the automated and manual approaches. For the automated approach, we used fifty bug reports, whereas, for the manual approach (3rd iteration), we used 383 bug reports. However, if we compare the outcomes of the automated and manual approaches, we can observe that the precision and recall for both are practically the same at about 65% and 76%, respectively. The outcome demonstrates that the final list of items we determined using the manual technique is precise enough to be used for the automated approach. Therefore, with the help of the final list of keywords and key-phrases, our regular expression-based `Plan Labller` is capable of achieving satisfactory accuracy.

Table 5.1: Precision, Recall and F1 Score of the Automated and Manual Approach

Approach	Title	Precision	Recall	F1 Score
Automated	Description	65.30%	71.11%	67.64%
	Comments	64.48%	79.31%	71.07%
	Total	64.47%	76.51%	69.97%
Manual	Description	65.56%	76.98%	70.81%
	Comments	64.64%	78.80%	71.02%
	Total	64.93%	77.68%	70.74%

5.3 Summary

In this chapter, first, we introduced a regular expression-based automated approach for implementing a `Plan Labeller`. Then, we presented the results of applying this implementation of the approach. Finally, we compared the results with our manual approach and found the results were almost identical with regard to precision and recall.

Chapter 6

Conclusion

In this work, we focused on identifying `Plan` in sentences of bug reports. The purpose of our study is to help users find the `Solution` or the `Suggestion` within the wide variety of information found in a bug report. This means that users or developers do not have to read each and every comment or description to find `Plans`. Our work sought to answer three research questions. First, we aimed to determine which sections of bug reports include `Plans`. To find the solution, we examine the Bug Title, Bug Description, and Bug Comments. From this, we found that developers express `Plan` using Bug Description and Bug Comments.

Next we analayzed the Bug Description and Bug Comments to determine whether any particular patterns, key-phrases, or keywords are used to express the `Plan`. We discovered that developers utilize a list of keywords and key-phrases to express `Plan` and that it is possible to identify `Plan` in sentences using these keywords and key-phrases. To determine the minimal list of keywords and key-phrases, we performed three iterations of applying the keywords and key-phrases to a set of bug reports and refined the keywords and key-phrases based on performance throughout each iteration. In the end, we examined 1,149 bug reports containing 13,565 sentences, which we manually labeled with `Plan` to derive a set of fifty-one keywords and key-phrases. We believe that our empirical results indicate that, in general, this set of keywords and key-phrases achieves sufficient precision and recall scores for practical use. This keyword and key-phrase set provides a good first step in the direction towards detecting `Plans` from sentences of bug reports, which will allow users to

create tag-based summaries more accurately.

6.1 Future Work

We have identified some future directions for this work.

During our study, we conducted three iterations to find the right set of keywords and analyzed how well they were able to detect `Plans` from the bug reports. The empirical results indicate that, in general, this set of keywords achieved sufficient precision and recall scores for practical use. However, doing more iterations may result in a better performing keyword set.

One of our goals in this work was to find out if it is possible to detect `Plans` from bug reports. We followed a manual approach where iterations, refining keywords, and even labeling were done manually. This approach is similar to that used by genetic algorithms. An investigation could be done that examines the results of using such algorithms to create the keyword set automatically.

As previously mentioned, Galappaththi et al. [11] in their work did not include the `Plan Labeller` to capture sentences with the `Plan` from bug reports. Their tag-based approach was lacking in providing a customizable summary with `Plan` to the users. Our keyword-based approach of detecting `Plans` could be integrated into Galapptahi et al.'s [11] tag-based approach.

Bibliography

[1] Hiteshwar Azad and Akshay Deepak. Query expansion techniques for information retrieval: a survey. 08 2017.

[2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery.

[3] Bird, Steven, Edward Loper, and Ewan Klein. Natural language processing with python: Natural language toolkit. https://www.nltk.org/#natural-language-toolkit, 2009. Accessed: 2022-10-09.

[4] Gerald Bortis and André van der Hoek. Porchlight: A tag-based approach to bug triaging. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 342–351, 2013.

[5] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 396–407, New York, NY, USA, 2017. Association for Computing Machinery.

[6] Indu Chawla and Sandeep K Singh. Automatic bug labeling using semantic information from lsi. In *2014 Seventh International Conference on Contemporary Computing (IC3)*, pages 376–381, 2014.

[7] Dictionary.com. The basic definition of plan with synonym and list of other idioms and phrases with plan. https://www.dictionary.com/browse/plan. Accessed: 2022-10-08.

[8] Joseph Downs. Card sorting: your complete guide towards card sorting approach. https://www.justinmind.com/blog/card-sorting/. Accessed: 2022-10-07.

[9] Jayalath Ekanayake. Bug severity prediction using keywords in imbalanced learning environment. *Int. J. Inf. Technol. Comput. Sci.(IJITCS)*, 13:53–60, 2021.

[10] Eibe Frank, Gordon W. Paynter, Ian H. Witten, Carl Gutwin, and Craig G. Nevill-Manning. Domain-specific keyphrase extraction. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, page 668–673, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[11] Akalanka Galappaththi and John Anvik. Automatic sentence annotation for more useful bug report summarization. In *MSc Thesis, Lethbridge, Alta.: University of Lethbridge, Department of Mathematics and Computer Science*, 2020.

[12] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[13] Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. Deep learning based valid bug reports determination and explanation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 184–194, 2020.

[14] Beibei Huai, Wenbo Li, Qiansheng Wu, and Meiling Wang. Mining intentions to improve bug report summarization. In *SEKE*, volume 2018, pages 320–363, 2018.

[15] Anette Hulth and Beáta Megyesi. A study on automatically extracted keywords in text categorization. In *ACL*, 2006.

[16] He Jiang, Najam Nazar, Jingxuan Zhang, Tao Zhang, and Zhilei Ren. Prst: A pagerank-based summarization technique for summarizing bug reports with duplicates. *International Journal of Software Engineering and Knowledge Engineering*, 27(06):869–896, 2017.

[17] Shubhra Goyal Jindal and Arvinder Kaur. Automatic keyword and sentence-based text summarization for software bug reports. *IEEE Access*, 8:65352–65370, 2020.

[18] Nivir Kanti-Singha Roy and Bruno Rossi. Towards an improvement of bug severity classification. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 269–276, 2014.

[19] Won Kim, Ok-Ran Jeong, and Sang-Won Lee. On social web sites. *Information Systems*, 35(2):215–236, 2010. Special Section: Context-Oriented Information Integration.

[20] Andrew Ko, Brad Myers, and Polo Chau. A linguistic analysis of how people describe software problems. pages 127–134, 01 2006.

[21] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 144–14411, 2018.

[22] Haoran Liu, Yue Yu, Shanshan Li, Yong Guo, Deze Wang, and Xiaoguang Mao. Bugsum: Deep context understanding for bug report summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 94–105, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Zhiyuan Liu, Chen Liang, and Maosong Sun. Topical word trigger model for keyphrase extraction. In *COLING*, 2012.

[24] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 430–439, 2012.

[25] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Softw. Engg.*, 20(2):516–548, apr 2015.

[26] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. Ausum: Approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[27] MongoDB. Pymongo: The official mongodb driver for synchronous python applications. https://www.mongodb.com/docs/drivers/pymongo/. Accessed: 2022-10-08.

[28] Ahmed Fawzi Otoom, Sara Al-jdaeh, and Maen Hammad. Automated classification of software bug reports. In *Proceedings of the 9th International Conference on Information Communication and Management*, ICICM 2019, page 17–21, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Fayola Peters, Thein Than Tun, Yijun Yu, and Bashar Nuseibeh. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering*, 45(6):615–631, 2019.

[30] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Summarizing software artifacts: A case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 505–514, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.

[32] Ali Sajedi-Badashian and Eleni Stroulia. Vocabulary and time based bug-assignment: A recommender system for open-source projects. *Software: Practice and Experience*, 50, 04 2020.

[33] Yang Song and Oscar Chaparro. Bee: A tool for structuring and analyzing bug reports. ESEC/FSE 2020, page 1551–1555, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Chengyu Sun, Liang Hu, Shuai Li, Tuohang Li, and Ling Chi. A review of unsupervised keyphrase extraction methods using within-collection resources. *Symmetry*, 12:1864, 11 2020.

[35] Xiaobing Sun, Wei Zhou, Bin Li, Zhen Ni, and Jinting Lu. Bug localization for version issues with defect patterns. *IEEE Access*, 7:18811–18820, 2019.

[36] Chengzhi Zhang, H. Wang, Y. Liu, Dan Wu, Y. Liao, and B. Wang. Automatic keyword extraction from documents using conditional random fields. 4:1169–1180, 06 2008.

[37] Yutong Zhao, Lu Xiao, Pouria Babvey, Lei Sun, Sunny Wong, Angel A. Martinez, and Xiao Wang. Automatically identifying performance issue reports with heuristic linguistic patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 964–975, New York, NY, USA, 2020. Association for Computing Machinery.

Appendix A

List of all Keywords

No	Type	List of Keywords	1st Iteration	2nd Iteration	3rd Iteration
1.	S	Actual	●	●	●
2.	S	Add	●		
3.	S	Agree	●	●	●
4.	P	Although	●	●	●
5.	S	Approach	●	●	●
6.	S	Attach	●	●	●
7.	S	Attempt	●	●	
8.	S/P	Because	●	●	●
9.	P	Bug	●	●	
10.	S	But	●		
11.	S	Change	●	●	●
12.	S	Check	●	●	
13.	S	Chose	●		
14.	S	Click	●		
15.	S	Close	●	●	
16.	S	Commit	●		
17.	S	Create	●	●	●
18.	S	Document	●	●	●
19.	S	Duplicate	●	●	●
20.	S	Easy	●	●	●
21.	S	Far	●	●	●
22.	S	Fix	●	●	●
23.	P	For Example	●	●	●
24.	S	Found	●	●	●
25.	P	Furthermore	●		
26.	S	Here	●		
27.	P	However	●	●	●

28.	S	I Think	•		•
29.	S	Impact	•		
30.	S	Implement	•	•	•
31.	S	Improve	•	•	•
32.	P	In Addition	•	•	•
33.	S	In Fact	•	•	•
34.	S	Include	•	•	•
35.	S/P	Instead	•	•	•
36.	S	Just	•	•	•
37.	S	Keep		•	•
38.	S	Like	•		
39.	S	Make	•	•	•
40.	S	Make Sure	•	•	
41.	S	Might		•	•
42.	S	Miss	•	•	•
43.	S	Modify	•		
44.	S	Need	•	•	•
45.	S/P	Option	•	•	•
46.	S/P	Other	•	•	•
47.	S/P	Otherwise	•	•	•
48.	S	Page	•		
49.	S	Patch	•	•	•
50.	S	Possible	•	•	•
51.	S	Probably	•	•	•
52.	S	Problem	•		

53.	S	Read	●	●	●
54.	S	Reasonable	●	●	●
55.	S	Resolve	●	●	●
56.	S	See	●		
57.	S	Seem	●	●	●
58.	S	Select	●	●	●
59.	S	Should/Would/Could+ Be	●		
60.	S	Show	●	●	
61.	S/P	Since	●	●	●
62.	S	Solution	●	●	●
63.	S	Solve	●	●	●
64.	S	Strategy	●		
65.	S	Success	●		
66.	S	Suggest	●	●	●
67.	S	Support	●	●	●
68.	S	Test	●		
69.	P	Therefore	●	●	●
70.	S	Update	●	●	
71.	S	Use	●		
72.	S	Want	●	●	●
73.	S	Whatever	●		
74.	S	Whether	●	●	
75.	S	Wonder		●	●
76.	S	Work	●	●	●
77.	S	You Can, Could/Can, Could You			●