



Run Run Shaw Library

香港城市大學
City University of Hong Kong

Copyright Warning

Use of this thesis/dissertation/project is for the purpose of private study or scholarly research only. ***Users must comply with the Copyright Ordinance.***

Anyone who consults this thesis/dissertation/project is understood to recognise that its copyright rests with its author and that no part of it may be reproduced without the author's prior written consent.

CITY UNIVERSITY OF HONG KONG
香港城市大學

Research on Methods in Locating
Software Features and Bugs
軟件特徵與缺陷的定位方法研究

Submitted to
Department of Computer Science
電腦科學系
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位
awarded by City University of Hong Kong
for successful completion of the joint programme with
Xi'an Jiaotong University

by

SHI Zhendong
施振東

July 2018

二零一八年七月

Abstract

of thesis entitled

Research on Methods in Locating Software Features and Bugs

by

Zhendong Shi

for the degree of Doctor of Philosophy

at City University of Hong Kong

in July 2018

Nowadays, software products are widely used in various fields. In the software development and maintenance processes, features from the requirements document and bugs from bug reports represent the wanted and unwanted functionalities of software projects. To implement these features or remove these bugs, finding the initial locations of the required features or buggy locations in the source code is important for developers. Particularly when the developers are the newcomers to a project, automatic localization approaches can help these new developers to quickly find the starting locations of features and bugs in an unfamiliar project. These approaches are generally referred to as feature location or bug localization. One of the main techniques used in feature location and bug localization is based on Information Retrieval (IR). This approach mainly calculates the textual similarity between source code entities and the feature description including bug reports, and then produces a ranked list of source code entities based on the similarities. Developers then examine the ranked list, beginning with the top entity, to find the target source entity, which is relevant to the feature or bug. Since feature location and bug localization have the same basic framework in IR-based approaches, bug localization is considered to be a specific area of feature location, and the bugs are considered to be unwanted features.

In this IR-based framework, there are several problems which limit the performance of location: (1) Recent automatic location methods do not consider the conventional methods. There is little research on the situation about the automatic method has low performance on bugs. (2) The existing methods of hybrid bug localization put little effort into the combining method and its influence on the localization performance. (3) There is little research work concerning the BM25 and BM25F algorithms in the information retrieval based feature location.

The three problems are distributed in three different parts of the whole location process. In order to increase the performance of feature location and bug localization, this thesis introduces the following three approaches to deal with the above problems:

1) This thesis proposes a bug localization strategy, which switches the localization process from the automatic ones to conventional ones, according to the prediction of the performance of both approaches in a current software project. Although some IR-based location techniques perform very well in some bugs, there are still some bugs hard to be localized using the automatic technique. For these bugs, conventional localization approaches may be more suitable than the automatic methods. The proposed strategy considers the situations including using automatic and conventional bug localization method. It optimizes the time point when developers switch the automatic method to the conventional method. This strategy increases the localization performance of the bugs which are hard to locate using automatic methods, but it has little influence on the well-performing bugs using automatic methods.

2) In hybrid bug localization methods, this thesis compares eight Learning to Rank (LtR) techniques to combine beneficial attributes from six different types and find the suitable integrating technique for hybrid bug localization. In recent decades, many additional attributes have been identified as being beneficial for bug localization. Attributes derived from the version history, stack traces, source code structure and so on, have been imported to localization approaches to help locate buggy source entities more precisely. However, the recent hybrid methods most use the linear combining method to integrate the beneficial attributes, or only use on LtR method. Little research puts effort on the influence of the combining method on bug localization performance. This thesis has researched the

performance of the eight LtR based bug localization methods and find that the coordinate ascent algorithm performs best in the selected attributes and data.

3) This thesis has research on BM25 and BM25F based feature location, which outperforms three conventional IR methods. BM25 and BM25F are popular information retrieval ranking algorithms, for which the performance is largely affected by their parameter settings. This thesis compares the feature location performance of BM25 and BM25F with that of three IR models, including Vector Space Model (VSM), Unigram Model (UM) and Latent Dirichlet Allocation (LDA), with different parameter values respectively. When applying the BM25F algorithm, the source code text is divided into two fields depending on whether the source entity is called. The results show that BM25 and BM25F are more effective than the three basic IR models for feature location.

To conclude, this thesis contributes three ways to increase the performance of IR based feature location and bug localization from three distinct aspects. This can help the software developers to quickly locate the source code entities which need to be modified, when receiving functional requirements and bug reports, thereby improving the efficiency of software development and reducing development and maintenance costs.

CITY UNIVERSITY OF HONG KONG
Qualifying Panel and Examination Panel

Surname: SHI

First Name: Zhendong

Degree: Doctor of Philosophy

College/Department: Department of Computer Science

The Qualifying Panel of the above student is composed of:

Supervisor(s)

Dr. KEUNG Wai Jacky	Department of Computer Science, City University of Hong Kong
Prof. SONG Qinbao	School of Electronics & Information Engineering, Xi'an Jiaotong University

Qualifying Panel Member(s)

Dr. YU Yuen Tak	Department of Computer Science, City University of Hong Kong
-----------------	---

This thesis has been examined and approved by the following examiners:

Dr. KEUNG Wai Jacky	Department of Computer Science, City University of Hong Kong
Prof. ZHANG Qingfu	Department of Computer Science, City University of Hong Kong
Dr. LI Minming	Department of Computer Science, City University of Hong Kong
Prof. ZHANG He	Software Institute, Nanjing University
Prof. QI Yong	Department of Computer Science and Technology, Xi'an Jiaotong University
Prof. YANG Xinyu	Department of Computer Science and Technology, Xi'an Jiaotong University
Prof. DONG Xiaoshe	Department of Computer Science and Technology, Xi'an Jiaotong University
Prof. LIU Jun	Department of Computer Science and Technology, Xi'an Jiaotong University
Prof. ZHANG Jing	Department of Computer Science and Technology, Xi'an University of Technology
Prof. WU Weiguo	Department of Computer Science and Technology, Xi'an Jiaotong University
Prof. DU Junzhao	Department of Software Engineering, Xidian University

Acknowledgments

I would like to express my deep gratitude to my supervisor Dr. Jacky Keung at City University of Hong Kong. Dr. Keung guided my academic research with patience and understanding, and constantly gave me great inspiration and much encouragement when I encountered problems in the process of my research. I feel very fortunate to have had such a good supervisor for my research.

I am deeply grateful to my supervisor Prof. Qinbao Song at Xi'an Jiaotong University. He was such a charming man, and as a professor, guided the direction of my research and enlightened my research efforts. Unfortunately, he passed away due to illness on Jan 17, 2016. His great scientific achievements and excellent teaching will never be forgotten.

I am also grateful to my new supervisor, Prof. Xinjun Zhang at Xi'an Jiaotong University, who gave me support and guided my research in the last period of my study at Xi'an Jiaotong University.

I am thankful to Dr. Y.T. Yu and Dr. W.K. Chan at City University of Hong Kong. They gave me valuable suggestions and comments when I encountered difficulties or problems in my research.

I also appreciate my classmates, co-authors, and roommates, in both Hong Kong and Xi'an, for their generous support for my research studies and life in Hong Kong. In particular, thanks go to Jinhua Jiang, my undergraduate classmate at Xi'an Jiaotong University, who participated in the City University of Hong Kong program, and Kwabena Ebo Bennin, who was my main partner in my research studies.

Last but not least, I would like to acknowledge my family for their generous love and endless support. Thank you all.

Contents

Abstract	i
Qualifying Panel and Examination Panel	iv
Acknowledgments	v
List of Figures	xi
List of Tables	xii
Chapter 1. Introduction	1
1.1. Overview	1
1.1.1. A Strategy to Determine When to Stop Using Automatic Bug Localization	2
1.1.2. Hybrid Bug Localization using Learning to Rank Techniques	3
1.1.3. BM25- and BM25F-Based Feature Location Techniques	3
1.2. Outline	3
Chapter 2. Background and Related Work	5
2.1. Information Retrieval	5
2.2. IR-Based Feature Location	6
2.2.1. Bug Report	8
2.2.2. Evaluation Metrics	9
2.2.3. Feature Location vs. Bug Localization vs. Fault Localization	11
2.3. Hybrid Methods of Bug Localization	12
2.3.1. Textual Information	15
2.3.2. Stack Trace	16
2.3.3. Version History	16
2.3.4. Dependence	17
2.3.5. Text Structure	18
2.3.6. Similar Reports	19
2.3.7. Execution	19

2.3.8. Discussion	20
2.4. Other Related Work	21
2.5. Summary	22
Chapter 3. A Strategy to Determine When to Stop Using Automatic Bug Localization	23
3.1. Introduction	23
3.2. Assumptions	25
3.2.1. Common Assumptions	26
3.2.2. Specific Assumptions	27
3.3. Probability of Being Buggy	28
3.3.1. Definition	29
3.3.2. Real Data	30
3.3.3. Classical Probability Model	32
3.4. Simulation of the Localization Process	34
3.4.1. Process Simulation	35
3.4.2. Optimization of ETC	36
3.5. Discussion	38
3.5.1. Comparison of using or not using the Strategy	38
3.5.2. Time Cost of the Conventional Approach	40
3.5.3. Another View of the Criterion	40
3.6. Threats to Validity	41
3.7. Summary	42
Chapter 4. Hybrid Bug Localization using Learning to Rank Techniques	43
4.1. Introduction	43
4.2. Learning to Rank Based bug Localization	44
4.2.1. Learning to Rank	46
4.3. Experimental Evaluation	47
4.3.1. Features	48

4.3.2.	Text Information	48
4.3.3.	Stack Trace.....	49
4.3.4.	Version History	49
4.3.5.	Dependence	49
4.3.6.	Text Structure.....	50
4.3.7.	Similar Report	50
4.4.	Dataset	51
4.5.	Tools.....	51
4.6.	Result.....	52
4.6.1.	Comparison to Reproduced Work.....	52
4.6.2.	SWT and ZXing Result.....	53
4.6.3.	Eclipse Result.....	54
4.6.4.	Result Comparison	56
4.7.	Threats to Validity	57
4.8.	Summary	57
Chapter 5.	BM25 and BM25F based Feature Location Techniques.....	59
5.1.	Introduction	59
5.2.	Related Work.....	60
5.3.	BM25-Based Feature Location.....	61
5.3.1.	BM25	62
5.3.2.	BM25F	63
5.4.	Case Study.....	64
5.4.1.	Data Set	65
5.4.2.	Preprocessing	65
5.4.3.	Setting	66
5.5.	Results and Analysis	68
5.6.	Threats to Validity	72
5.7.	Summary	73

Chapter 6. Conclusions and Future Work	75
6.1. Conclusions	75
6.2. Future work	76
Bibliography	78
List of publication	84

List of Figures

Figure 2.1 A general framework for IR-based feature location	7
Figure 2.2 Bug #120704 from the Eclipse project	9
Figure 2.3 An example of a stack trace in bug report #110354 from Eclipse.....	16
Figure 3.1 Number of buggy entities in the ranked list: Eclipse.....	31
Figure 3.2 Number of buggy entities in the ranked list: SWT	32
Figure 3.3 P_k for each rank: Eclipse.....	34
Figure 3.4 P_k for each rank: SWT	34
Figure 3.5 Average number of entities that need to be examined to fix a bug for different values of X: Eclipse ($T_b=20T$).....	39
Figure 3.6 Average number of entities needing to be examined to fix a bug for different values of X: SWT ($T_b=10T$).....	40
Figure 4.1 The flowchart of the LtR based hybrid bug localization	45
Figure 4.2 The MAP values in folds of Eclipse project.....	55
Figure 5.1 MRR for IR techniques with best configurations	72

List of Tables

Table 2.1 Input features in hybrid approaches to bug localization	13
Table 2.2 Summary of hybrid methods in bug localization.....	14
Table 3.1 Number of buggy entities in the top 10 of the ranked lists of all bugs for each project	30
Table 3.2 Probability P_k for the top 10 entities in the ranked list	33
Table 4.1 The three group approaches of LtR algorithms	47
Table 4.2 Temporal attributes of input data types	48
Table 4.3 Details of BugLocator dataset.....	51
Table 4.4 The features used in BLUiR, AmaLgam and our LtR approach	52
Table 4.5 Average MAP result of SWT project in 5-fold cross-validation	53
Table 4.6 Average MAP result of ZXing project in 5-fold cross-validation	54
Table 4.7 MAP comparison of LtR based approach and reproduced state-of-art techniques	56
Table 5.1 Two-field BM25F parameters used in feature location	64
Table 5.2 Metrics of software systems	65
Table 5.3 LDA parameter settings	68
Table 5.4 BM25 parameter settings	68
Table 5.5 BM25F parameter settings.....	68
Table 5.6 The best and worst four configurations for VSM, UM, and LDA for feature location.....	70
Table 5.7 The best and worst four configurations for BM25 and BM25F for feature location.....	71

Chapter 1. Introduction

In recent decades, computer software has been widely applied in numerous fields of human society. The size and complexity of software are constantly increasing. To reduce the cost of software and improve the efficiency of its development, various automatic methods for feature location and bug localization have been proposed in software engineering.

Feature location is a software comprehension activity that aims to identify source code entities that implement functionalities. If these functionalities are unwanted, such as bugs, this location task is also known as bug localization [1]. Bug localization involves finding buggy source code entities, which then have to be modified to correct the bug [2].

This thesis focuses mainly on feature location and bug localization based on information retrieval (IR). IR involves finding a related document based on a query. Both feature location and bug localization use IR techniques to compare the textual similarity of the query (the bug report/feature) and the text of source code entities (the class/method). In addition to IR-based techniques, dynamic methods (also called spectrum-based methods) are used in this area. Dynamic methods primarily utilize execution traces of the designed scenarios to calculate the relatedness of source code entities.

This thesis contains three main parts, each of which aims to increase the performance of feature location and bug localization:

- 1) A strategy is proposed for optimizing the time point of switching from the automatic bug localization methods to conventional ones;
- 2) Hybrid bug localization methods are studied using learning to rank (LtR) techniques, and eight different LtR methods are compared;
- 3) The BM25 and BM25F techniques are studied and applied to feature location.

1.1. Overview

In feature location, the ‘features’ refer to software functionalities or requirements that need to be located in the source code; when these have been found, developers can add new features to the project, improve existing functionalities, or fix bugs in the source

code. In routine software maintenance, bug reports are submitted by users when they encounter bugs in software. In order to fix each bug reported, the buggy location in the source code first needs to be found. The goal of bug localization is to find those source code entities that need to be modified to fix the bug, according to the bug reports [2].

Feature location typically refers to the process of finding locations in the source code that correspond to specific functionalities (features) [1]. Some researchers treat bug localization as a sub-area of feature location [1], [3]; feature location can be called bug localization when all features are bugs, and when bugs are treated as unwanted features of software systems. Some researchers [4], [5] consider feature location as a different area from bug localization, based on the idea that features do not contain bugs. In this view, features only contain the aspects or concepts that developers want to implement.

This thesis mainly focuses on IR-based feature location and bug localization. We introduce three methods to improve the performance of finding related source code entities. These three approaches are summarized in the following subsections.

1.1.1. A Strategy to Determine When to Stop Using Automatic Bug Localization

IR-based automatic bug localization techniques provide developers with a list of suspicious buggy source entities, with the aim of finding the one that needs to be modified to fix the bug. However, it is unavoidable that some buggy entities are ranked low in the result list when using these automatic techniques. Due to this limitation, it is therefore important for developers to understand when to stop using the ranked list and to use a conventional bug localization approach instead. This study introduces a novel strategy for addressing this challenge, based on the ranked list of suspicious buggy entities [6]. We assume a bug localization process in which after each source code entity in the ranked list has been examined, the developers have the option of whether to continue examining the automatic bug localization results or to switch to the use of a conventional localization approach. We propose a new evaluation metric called the expected time cost (ETC) that we want to minimize, which includes the time cost of using both automatic and conventional approaches. Under these assumptions, we derive simple criteria to minimize the ETC. We compare the time cost of a state-of-the-art automatic localization method called BugLocator [7], both with and without the use of our strategy, for two projects. The results show that the use of our strategy can

significantly reduce the time cost compared to using the automatic localization technique alone and using the conventional approach alone.

1.1.2. Hybrid Bug Localization using Learning to Rank Techniques

As well as IR-based similarities, features extracted from version history [7]–[9], dynamic analysis [10]–[12], source code structure [13]–[15] and other sources have also been found to be beneficial for bug localization. We conduct a short survey of hybrid bug localization methods that use additional features besides IR similarity, and then compare eight different LtR techniques with hybrid bug localization methods that combine several beneficial features to improve the precision of localization. LtR involves the application of machine learning to ranking models for IR [16]. The experimental results show that a coordinate ascent algorithm without normalization is a suitable LtR technique for use in bug localization for selected attributes, outperforming two state-of-the-art localization approaches in two large projects.

1.1.3. BM25- and BM25F-Based Feature Location Techniques

Manual feature location is a labor-intensive task, requiring developers to find target entities from thousands of software artifacts. Recent research has developed automatic and semi-automatic methods, primarily based on IR techniques, for locating source entities that are textually similar to a given feature. In this study, we focus on individual IR-based methods with the aim of finding a suitable IR technique for feature location that could be used as a part of a hybrid method, in order to achieve good performance. We present two feature location approaches based on the BM25 algorithm and its variant, BM25F [17]. We compared these two algorithms to the vector space model (VSM), the unigram model (UM) and latent Dirichlet allocation (LDA) for four open source projects. The experimental results show that BM25 and BM25F are consistently better than other IR methods, including VSM, UM, and LDA, for the four selected software systems in their best configurations.

1.2. Outline

The basic principles and background of IR-based feature location, hybrid bug localization and LtR and prior work related to these topics are introduced in Chapter 2. The following three chapters present three ways of increasing the performance of feature location and bug localization, as follows.

In Chapter 3, we propose a switching strategy between automatic and conventional bug localization techniques. This strategy largely solves the problem of automatic techniques showing low performance for some bug reports. A new evaluation metric called ETC is proposed, which includes the costs of both conventional and automatic methods.

In Chapter 4, we introduce an approach that uses LtR techniques to combine various input data that have been shown to offer benefits for bug localization, in order to increase the precision of bug localization. We also compare eight different LtR algorithms and their performance in hybrid bug localization. The experimental results show that the coordinate ascent algorithm without normalization performs better than the other LtR algorithms.

In Chapter 5, we present feature location approaches based on BM25 and its variant, BM25F. BM25 and BM25F are two popular ranking algorithms for IR. The experimental results show that BM25 and BM25F achieve better performance than VSM, UM, and LDA for feature location.

Conclusions and future work are discussed in Chapter 6.

Chapter 2. Background and Related Work

In this chapter, we introduce the background to IR-based feature location and bug localization, and prior work related to the three methods proposed here.

2.1. Information Retrieval

IR is the study of searching for documents based on a text query [18]. In recent years, search engines such as Google and Yahoo have mainly used IR technology to help users find web pages in which they are interested. The input query of an IR system is usually a short text; the system then retrieves all candidate documents and finally outputs a list of documents, ranked according to the relevance between the query and the document. The higher a document is ranked, the more relevant it is to the query.

In the IR system, documents and queries are first preprocessed using natural language processing (NLP), which includes the removal of stop words and stemming. Stop words that need to be removed from the collection of documents include a list of valueless words in a language, such as “to”, “the”, “do”, and so on in English. When an IR algorithm is applied to the software source code, the stop words also include the keywords of the programming language, such as “public”, “abstract” and “while” in Java, and some common class and method names, such as “java”, “system”, “util” (also in Java). A stemmer, as used by Porter [19], derives root words from terms in documents, transforming words such as “opening”, “opens”, “opened”, into their common root word “open”. In this way, similar terms using different tenses and variants can be recognized as having the same meaning.

Following the application of NLP, the documents and queries are represented in suitable models such as VSM [20], the smoothed unigram model (SUM) [21], latent semantic indexing (LSI) [22] and LDA [23]. A brief introduction to these representation models is given below.

VSM: A document or a query is represented as a term-weighted vector, typically using a ‘term frequency-inverse document frequency’ (*tf-idf*) scheme.

UM: A unigram model is a probabilistic model in which the documents are ranked based on their probabilities of containing all the terms of the query in each document. This model is usually smoothed to avoid a zero probability (in this case, the document does not contain all the terms existing in the query). This is usually called SUM.

LSI: LSI uses singular value decomposition (SVD) to produce a low-dimensional space to represent documents from term-document matrices. LSI is also known as LSA (latent semantic analysis).

LDA: LDA is a probabilistic topic model which introduces a topic layer between the document and the words. Each topic is represented as a vector of term features, and each document is represented as a vector of the topic feature.

Unlike SUM, which provides the ranking scheme for the model, the other three models provide only a way of representing documents. The ranked documents list is produced by applying similarity measures such as the cosine similarity [18].

2.2. IR-Based Feature Location

In this sub-section, we will describe how IR techniques are applied in feature location. A general framework for the IR method as used in feature location is shown in Figure . Software features such as bug reports are the inputs in this framework. The goal of this IR system is to find source code entities that are relevant to a given feature. Based on the different granularity settings, the retrieval of the source code entities may be at the level of methods/functions, classes or files. In this framework, the source code entities are treated as documents in the search engine, and a document may therefore be a method/function, a class or a file, depending on the predetermined granularity. The output of this framework is a ranked list of source code entities (documents). The higher buggy source code entities are ranked, the better performance of feature location is.

In an intuitive way, the text description of the feature contains semantic information about the functionalities, and the source code identifiers are also named for their functionalities. Comments in the source code may also contain relevant semantic information about the feature. Hence, the textual information in the relevant source code entity tends to be similar to the feature description, and IR techniques can provide various types of text similarity calculation methods to locate these features.

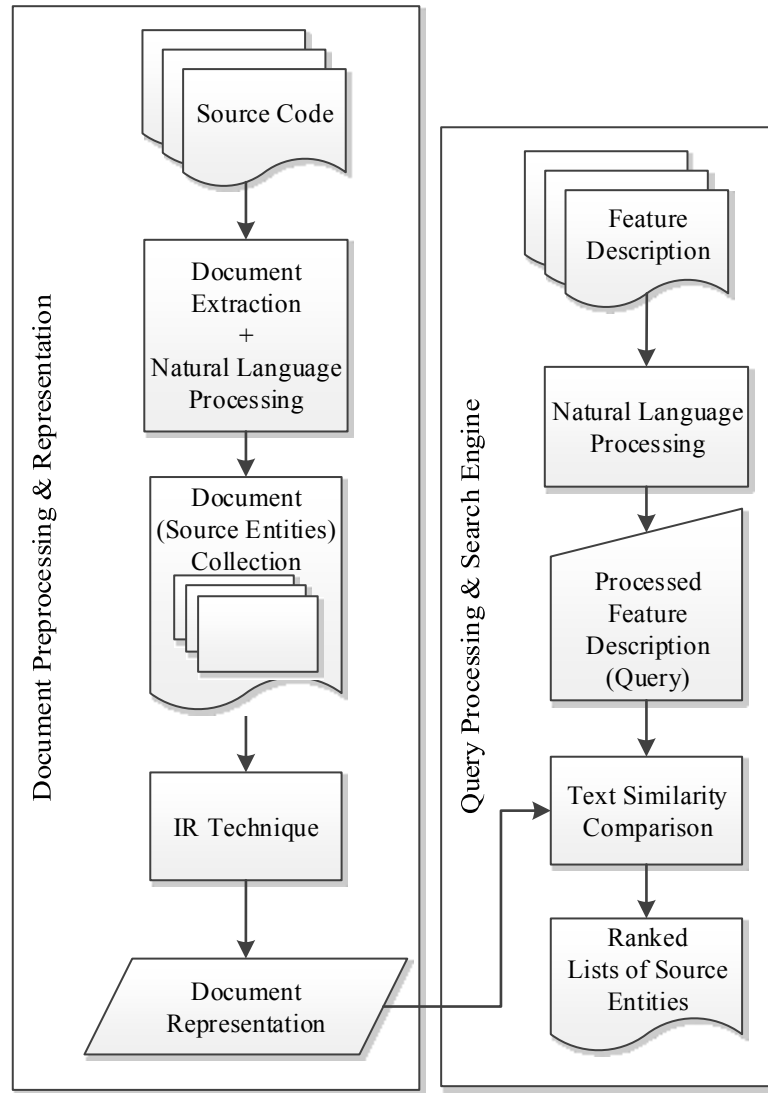


Figure 2.1 A general framework for IR-based feature location

The framework in Figure 2.1 can be divided into two main parts. The first of these involves document preprocessing and representation, and is shown on the left-hand side of Figure 2.1. After the source code is input, the source code terms are first extracted from each document; this usually means that the terms in the identifiers and comments will be extracted. Then NLP steps are applied to the terms extracted from each document, including the removal of stop words, splitting of multi-words and stemming. The stop words that need to be removed include a general list of English stop words, keywords related to the programming language, and punctuation. Multi-word identifiers (for example, “OpenFile”) are split into single terms (such as “open” and “file”) using techniques such as CamelCase [24]. Following this, a stemmer is applied to the terms, as used by Porter [19], to obtain the root words. The effects of choosing only identifiers

or comments, and the choice of different combinations of preprocessing steps, have been researched [25].

After preprocessing the raw text from the source code, the documents derived from the source code entities are usually represented as term frequency vectors, i.e. as ‘bags of words’. This model assumes the text has no ordering of the words [18]. All textual methods mentioned here are based on this assumption. The document representation vectors differ based on the different IR techniques used; for example, one document representation used in VSM is *tf-idf*, which considers the number of times a term occurs in a document and the number of documents including this term. In the LDA model [23], each document is represented by a document-topic probability distribution, while each topic is represented by a topic-term probability distribution.

In the right-hand side of the framework shown in Figure 2.1, the feature descriptions are first preprocessed using the same procedures as for the source code documents. The preprocessed feature descriptions are then input into the search engine as queries. The similarity between a query and a document depends on which IR technique is used. This IR system utilizes the similarity between the text in the source code elements and the textual description of the feature in order to generate a ranked list of documents. Developers are expected to be able to find the target entity quickly based on the ranked list.

When the features are in fact bugs (unwanted features), feature location is referred to as bug localization [1]. The aim of bug localization is to find those source code entities which need to be modified to fix the bug. The feature description contains the text information in the bug report, which includes the abstract, summary and comments regarding the bug. Some researchers consider bug localization to be different from feature location, and this is discussed further in Section 2.2.3.

2.2.1. Bug Report

Bug localization is considered to be a sub-area of feature location, and a bug report as a type of feature. To find the location of the buggy parts of the source code, static and dynamic methods are used in bug localization. In static methods, a bug report is taken as the input, and a ranked list of potential buggy source entities (files/methods) are returned. A bug report is typically written and submitted by a software user, and describes a bug they have encountered. An example of a bug report from the Eclipse

project is shown in Figure 2.2, and can be downloaded from Eclipse's Bugzilla¹ website. A bug report generally contains a bug ID, an open date (the date the bug was submitted), a fix date (the date the bug was fixed), a summary, and a description (detailed information about the bug). Fixed files are extracted from the version control system by searching for text such as "Fixed 120704" or "bug #120704" in log messages [26]. IR techniques use the summary and description text as the query and search source code for textually similar files. In a similar way to web search engines [27], bug reports resemble input queries and source code files resemble candidate web pages.

Bug ID	120704
Open date	2005-12-13 15:14:00
Fix date	2006-08-09 13:05:00
Summary	[Wizards] Problems with the "New" wizard
Description	<ol style="list-style-type: none"> 1. There is a small red square in the top right. 2. The filter does not line up on the right with the viewer and looks bad. 3. The viewer is small vertically, and yet there is a lot of real estate that is being unused on the bottom. 4. Why is there a help icon on the bottom right, and why is it never enabled? <p>Screenshot to follow.</p>
Fixed files	org.eclipse.ui.internal.dialogs.NewWizardNewPage.java

Figure 2.2 Bug #120704 from the Eclipse project

2.2.2. Evaluation Metrics

Top-N, mean average precision (MAP) and mean reciprocal rank (MRR) are the three main evaluation metrics used to measure the effectiveness of an IR system. These metrics are widely used in IR-based bug localization [4], [5], [7], since bug localization uses the same framework as IR, that is, using textual similarities to produce a ranked list of source code entities (documents in IR) for a given bug report (query). Lines Of Code (LOC) is an evaluation metric that is only used in the software retrieval domain. A brief summary of these evaluation metrics is given below.

¹ https://bugs.eclipse.org/bugs/show_bug.cgi?id=120704

Top-N refers to the number of bugs with related source code entities ranked in the top N entities (N=1, 5, 10) of the result list. For a bug report, if one of the entities which needs to be modified to fix the bug appears in the top N items of the ranked list, the bug report is considered to be localized, and is counted in the Top-N result. The higher the value of Top-N, the better the performance of the bug localization method.

MAP provides a single-figure measure of IR quality [18] when a query has multiple related documents. Average precision (AvgP) is the average value of the precision for a single query $q_j \in Q$, defined as follows:

$$AvgP_j = \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk}) \quad (2.1)$$

where the set of related documents for query q_j is defined as $\{d_1, \dots, d_{m_j}\}$; m_j is the number of related documents of query q_j ; and R_{jk} is the set of ranked retrieval results from the top result until document d_k is obtained. The precision of R_{jk} is defined as follows (where the symbol # indicates “the number of”):

$$Precision(R_{jk}) = \frac{\#(related\ documents\ in\ R_{jk})}{\#(documents\ in\ R_{jk})} \quad (2.2)$$

Mean average precision is the mean of AvgP over all queries:

$$MAP = \frac{1}{|Q|} \sum_{j=1}^{|Q|} AvgP_j \quad (2.3)$$

where $|Q|$ is the number of queries. A higher value of MAP indicates a better performance.

MRR is a statistical measure depending on the first relevant document in the ranked list for a query [28]. In bug localization, the reciprocal rank of a query is the multiplicative inverse of the effective measure, where the effective measure (EM) is the highest rank of the relevant source code entities for the bug in the final ranked list [12]. MRR is the mean of the reciprocal rank for all queries:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{EM_i} \quad (2.4)$$

The higher the value of MRR, the better the method performs.

LOC is used in the field of software retrieval. In bug localization, it indicates the lines of code which need to be examined to localize the bug. It is used only rarely in the area of IR-based bug localization, compared to the previous three metrics.

In addition to the standard evaluated metrics used in IR, some researchers have studied the question of how to evaluate the localization task in a more precise way. Beard et al. [29] discussed the question of whether locating the top buggy method in the final list enables the user to find the complete set of buggy methods. These authors claimed that the first method returned in the ranked list of documents relevant to the bug can be extended to locate 60% of the full set of relevant methods, but that this is not sufficient to find all relevant methods.

Hill et al. [30] proposed a rank topology metric for feature location, and assumed that two feature location techniques, A and B, are used. If A and B both have their first relevant entity ranked in fifth place, but the first four entities returned for A in the list are all irrelevant, while B's first four returned entities all need to be understood to locate the feature, then, intuitively, B is better than A. In contrast, measures such as MAP, MRR would report A and B as performing equally. This proposed metric is used to find the differences between the results of techniques such as A and B, and to give B a higher score.

Tantithamthavorn et al. [31] investigated the impact of the levels of granularity on feature location techniques, and presented a new feature location evaluation metric based on LOC, which the developers checked using the location task. They found that feature location at function level was more efficient than at class level.

2.2.3. Feature Location vs. Bug Localization vs. Fault Localization

Feature, bug and fault localization are three widely used phrases in software engineering, although they are used with different meanings by different researchers. Here, we will discuss the different definitions of these three concepts.

Feature location and bug localization are retrieval activities in software engineering. The aim of feature location is to help developers to quickly find which part of the source code needs to be added or modified to implement the required functionality [1]. Generalized feature location is a broad area including concept, bug and fault location/localization. Some researchers [1], [3] approach feature location using this generalized concept, and consider bug localization to be a sub-area of feature location

in which bugs are treated as undesired features that need to be removed. These feature location techniques use bug reports as features in case studies, since feature descriptions are usually hard to obtain from the requirements document. Other researchers [4], [5], [25], [32] consider feature location as a narrower concept, and treat bug localization as a different area from feature location. They consider that the features in feature location include only requirement functionalities rather than bugs.

Wong et al. [32] used bug-oriented fault localization as a synonym for bug localization. However, fault localization usually implies dynamic fault localization at the statement level for a test case [33], [34], and is treated as dynamic feature location in the generalized concept of feature location [1]. Since these concepts do not yet have a unified definition, it is hard to distinguish whether a static or dynamic method is used solely by looking at the concept terms (fault localization, bug localization and so on), although it can be determined whether this is a spectrum-based dynamic method or an IR-based static method based on the contents in their papers [13].

2.3. Hybrid Methods of Bug Localization

The primary technique of IR-based bug localization involves the use of the textual similarity between bug reports and source entities to find relevant buggy source entities. Recent research in bug localization primarily focuses on identifying additional features such as version history [5], [9], source code structure [13]–[15] and dynamic analysis [10]–[12] that could be utilized in ranking the relevant source code entities. These approaches usually import one or more new features for IR-based bug localization, in order to improve performance. One trend utilizes additional information to help locate bugs, since growing numbers of useful features besides textual similarity are found to offer benefits for the efficiency of bug localization [5]. Approaches that involve the use of features other than textual similarity in bug localization are called hybrid bug localization; conventional IR-based bug localization uses only textual similarity.

Seven main types of feature are used in hybrid bug localization approaches, and a summary of these features is shown in Table 2.1. For each feature in this table, a simple description and an explanation of how this feature can help to locate bugs are presented. Recent hybrid bug localization methods and the type of features they use are shown in Table 2.2. These hybrid methods typically use two or three features to enhance the location procedure, while the main feature is the textual similarity between the source

code entities and bug reports. We describe the task phrase (feature/bug/fault location/localization) used by the researchers in Table 2.2.

Table 2.1 Input features in hybrid approaches to bug localization

Input data	Description	Usage for bug localization
Text information	Textual information of bug reports and source code entities.	Source code entities that are textually similar to the query tend to be related to the bug [35].
Stack trace	Some bug reports contain stack traces when the software crashes.	Entities in the stack are likely to be buggy [32], [36], and especially the entity in the first frame of the stack trace.
Version history	This refers to records of the revision history of source code entities.	Revised entities in recent versions are more likely to be buggy [5], [9]. Bugs tend to reoccur in the same location.
Dependence	This refers to the dependence graph of source code methods and classes.	Buggy entities tend to have special ranges of web scores, such as PageRank and HITS [10].
Text structure	Terms in the source code can be classified into various fields, such as the class name, method name, variable name, comments, and so on. Bug reports contain title and description fields.	Different fields have different degrees of importance; for example, class names are more likely to have a vocabulary in common with bug reports than other code locations [37].
Similar report	This refers to previous fixed bug reports that are textually similar to the current bug report, and their relevant source entities.	Buggy methods related to previous similar bug reports tend to be buggy for the current report [3], [5]. Similar errors often have the same underlying reason.
Execution	This refers to execution traces from testing or debugging the software.	Entities in execution traces that trigger the bug, but which do not occur in non-bug execution traces, are likely to be related to the bug [11], [12].

Table 2.2 Summary of hybrid methods in bug localization

Author ‘Year	Approach name	Feature/Bug/faUlt	Input data							Granularity		Combining Method		
			Text information	Stack trace	Version history	Dependence	Text structure	Similar report	Execution	Method /Function	Class/file	Linear	Filter	other
Liu’07 [11]	SITIR	F	•	•	•	•	•	•	•*	•	•	•	•	•
Poshyvanyk’07 [12]	PROMESIR	F	•	•	•	•	•	•	•*	•	•	•	•	•
Nichols’10 [38]	•	B	•	•	•	•	•	•*	•	•	•	•	•	• ¹
Rao’11 [39]	•	B	•* ²	•	•	•	•	•	•	•	•	•	•	•
Nguyen’11 [40]	BugScout	B	•	•	•	•	•	•*	•	•	•	•	•	• ³
Zhou’12 [7]	BugLocator	B	•	•	•	•	•	•*	•	•	•	•	•	•
Sisman’12 [9]	•	B	•	•	•*	•	•	•	•	•	•	•	•	• ⁴
Davies’12 [3]	•	B	•	•	•	•	•	•*	•	•	•	•	•	•
Lal’12 [41]	•	U	•	•	•	•	•	•	•	•	•	•	•	•
Saha’13 [42]	BLUiR	B	•	•	•	•	•*	•	•	•	•	•	•	•
Tantithamthavorn’13 [43]	•	B	•	•	•*	•	•	•	•	•	•	•	•	•
Dit’13 [10]	•	F	•	•	•	•*	•	•	•	•	•	•	•	•
Moreno’14 [44]	Lobster	B	•	•	•	•*	•	•	•	•	•	•	•	•
Wang’14 [5]	AmaLgam	B	•	•	•*	•	•	•	•	•	•	•	•	•
Wong’14 [32]	BRTracer	U	•	•*	•	•	•	•	•	•	•	•	•	•
Ye’14 [45]	•	B	•*	•	•	•	•	•	•	•	•	• ⁵	•	•
Le’15 [46]	AML	B	•	•	•	•	•	•	•*	•	•	• ⁶	•	•
Dilshener’16 [47]	•	B	•	•	•	•	•*	•	•	•	•	•	•	•
Wang’16 [48]	AmaLgam+	B	•	•	•*	•	•	•	•	•	•	• ⁷	•	•
Ye’16 [49]	•	B	•*	•	•	•	•	•	•	•	•	• ⁵	•	•
Youn’17 [50]	BLIA	B	•	•	•	•	•	•*	•	•	•	•	•	•

¹ Textual terms from old bug reports are added to related historical buggy methods.² This approach combines similarity scores from various IR techniques, including UM, VSM, LSA, LDA and the cluster-based document model (CBDM).³ The entire algorithm is an integrated topic model.⁴ This combination of techniques used Bayesian probability, where the prior probability is drawn from the version history, the likelihood is derived from the IR technique, and the posterior probability is the final score.⁵ These two approaches use SVM^{Rank} [51] to learn linear weight values.

⁶ AML uses a probabilistic learning approach to train the linear weights.

⁷ AmaLgam+ uses the genetic algorithm (GA) to tune the weights of linear combination by analyzing fixed bugs.

Twenty-one hybrid approaches to bug location are presented Table 2.2, and seven types of input features are discussed in detail in the following section. These hybrid methods are introduced in the subsections indicated with a star (*) in Table 2.2. The star (*) indicates the summary location of the corresponding paper within Section 2.3, as shown in the Input data column. For example, for Liu'07 [11] in the first row of the table, •* is shown in the Execution sub-column of the Input data column. This indicates that a summary of this paper is given in the Execution subsection of Section 2.3.

2.3.1. Textual Information

IR techniques have been widely used in report-oriented bug localization. A technique called BLUiR [4] and previous work by the current author [17] have both shown BM25 to have good performance when used in bug localization. BM25 [52] is a bag-of-words ranking function, implemented in the Okapi IR system. Each query term has a score that depends on the occurrence of this term in all documents, regardless of the interrelationship between the query terms within a document.

Rao et al. [39] compared five text models for use in feature location: UM, VSM [18], LSA [22], LDA [23], CBDM and combined similarity scores from these IR techniques. The results indicate that sophisticated models such as LDA, LSA and CBDM perform worse than simple models such as UM and VSM. For example, in a linear combination model based on LDA and UM, a reduction in performance was observed as the weight of the LDA increased. Thomas et al. [25] studied the effects of different configurations of VSM, LSI and LDA on bug localization, and found that the configuration is important, both in the individual model and in the proposed combination framework.

Ye et al. [45] used an LtR algorithm called SVMRank [51] to find files relevant to bug reports. The authors extracted a new dataset using the automatic method provided by iBugs [26]. Features used in addition to IR similarity include the lexical similarity of APIs, bug-fixing recency, collaborative filtering score and others. All attributes are normalized using feature scaling in these techniques. More features are used in the new version of this approach [49], which outperforms the previous one. In total, 19 features are used that were not used in the previous version of this approach, including the

structured text information in BLUiR [4], webpage link scores including HITS [53] and PageRank [54] scores.

2.3.2. Stack Trace

Sometimes, a bug report contains a stack trace for the failing thread, as shown in Figure 2.3. A stack trace shows the sequence of instructions that were executed before a crash. These can also be considered in a ranked list of source code files that are potentially relevant to the bug. The intuitive concept behind the use of stack traces is that a file with higher rank in the stack trace is given a higher score, meaning that this file is more likely to be buggy.

```
!ENTRY org.eclipse.ui 4 0 2005-09-22 14:09:47.664
!MESSAGE java.lang.NullPointerException
!STACK 0
java.lang.NullPointerException
at org.eclipse.ui.internal.PartSashContainer.drag(PartSashContainer.java :1034)
at org.eclipse.ui.internal.dnd.DragUtil.getDropTarget(DragUtil.java:338)
...
at org.eclipse.core.launcher.Main.main(Main.java:952)
```

Figure 2.3 An example of a stack trace in bug report #110354 from Eclipse

Schroter et al. [36] found that almost 60% of fixed bug reports with stack traces could be fixed by modifying the methods shown in the stack traces. The authors note that 40% of the modified methods could be found in the first frame of the stack trace, and almost 90% of the modified methods could be found in the top 10 frames of their stack traces.

Wong et al. [32] added source code segmentation to filter noise, based on BugLocator [7]. This technique segments a file into several parts, compares the similarity between the bug report and each of these parts, and uses the one with the highest similarity. The authors also used a stack trace feature to increase the performance of their technique. Thung et al. [55] proposed an integrated tool called BugLocalizer, based on the BugLocator algorithm, which uses previously fixed bugs.

2.3.3. Version History

The intuitive basis for the use of the version history component is as follows: if a method was implemented or modified only in recent versions, it has a large probability of causing bugs, since bugs are usually created by new functionalities or buggy partitions that have been newly modified.

Sisman et al. [9] used the version history to estimate the probability that an existing file will be the target of bug localization. If a file is modified to fix a bug, or if a feature has

been implemented many times in the version history, it is very likely to be the source of similar bugs or features. An important issue is how to weigh the score from IR techniques and historical data. The author attempted several methods of normalization in order to obtain the best result.

Tantithamthavorn et al. [43] improved the BugLocator method by considering co-change histories of the source code. These authors leverage the assumption that when a buggy file is fixed, then any other files modified in the same commit should also be responsible for the bug. This approach creates a co-change matrix for files, which records the number of pairs of files that were modified simultaneously in the history. The values in the matrix are normalized using feature scaling. A file is added to the top N list if the co-change value is higher than a given threshold and if the co-changed file is also in the top N list.

AmaLgam [5] combined version history, similar reports, and source code structure to give improvements in bug localization. It provided a modified function based on Google's approach [56] toward the version history component, which uses the period between the previous modification time of the file and the time of input of the bug report. This function is introduced in Section 4.3.4, since it is used in the experiments in the current work. In addition to the version history component, a component of AmaLgam based on similar reports is derived from BugLocator [7], while the source code structure of AmaLgam is drawn from BLUiR [4]. AmaLgam+ [48] is an extended version of AmaLgam in which two components (the stack trace and reporter information) are added to the AmaLgam approach. The stack trace score of a file is simply the reciprocal of the file rank in the stack trace, as used in [32]. The intuitive basis for the use of the reporter information component is that a given bug reporter is likely to report issues from the same or similar software components.

2.3.4. Dependence

Dit et al. [10] applied the idea of data fusion to feature location. Their approach combined IR (using LSI), execution (using SITIR [11]), and link analysis (using HITS [53] and PageRank [54]) to improve the performance of feature location, constructing a dependence graph of source code methods such as web pages and examining the relationship between the PageRank and HITS scores of these methods to determine whether they were buggy. The researchers concluded that filtering methods based on the lowest hub scores perform best when combined with other information; this is because

methods with low hub scores usually have few connections to others, and are more similar to getter and setter methods, which are less likely to be buggy.

Moreno et al. [44] proposed Lobster, which combines stack traces in bug reports with textual bug localization. The authors found that a shorter distance between the currently checked source entity and the entities in the stack trace indicated a higher probability of the current source entity containing bugs. The distance between two source code entities is calculated based on the dependence graph [57]. The structure similarity is obtained based on the distance between the current source entity and the stack trace in the bug report, and the final similarity score is a linear combination of the textual similarity and structure similarity.

2.3.5. Text Structure

BLUiR [4] divides the source code files into four fields (class, method, variable, and comment) and bug reports into two fields (summary and description), and then calculates the textual similarity for each pair of fields from source code files and bug reports. Lastly, this technique uses the sum of various attributes as a probability score for the text similarity. In our approach, we use eight attributes to indicate the eight different field-pair similarities from BLUiR, and leave the remainder of the learning procedure to the LtR algorithm.

Lal and Sureka [41] presented a technique that used a character n-gram IR method to enhance the similarity between bug reports and related source code entities with the same filename or path. In this case, the file/path names in the bug report are considered important fields that need to be enhanced to obtain improved localization performance.

Dilshener et al. [47] proposed a novel bug localization method without using past project information such as similar bug reports and version history components, but which outperformed algorithms such as BugLocator [7], BLUiR [4] and AmaLgam [5], which utilize past project information. This method places higher importance on file names; files whose name shares a common word in a key position of the bug report summary are given a higher relevance score. The authors also used the stack trace to increase localization performance, and placed higher importance on file names in terms of textual similarity. These ideas built upon those of other researchers, such as the use of stack traces [44] and the separation of a bug summary from its description [4], and the authors also carried out a manual analysis of the source code files and bug reports. For example,

for the SWT project, they found that the first term of the report summary was affected by the source file in 42% of bug reports, while the fourth term of the report summary was affected by the source file in only 15% of bug reports.

2.3.6. Similar Reports

Nichols [38] added terms from old bug reports to the relevant buggy methods. In this way, the textual information of the methods was given an additional part representing the relevant fixed bugs. BugScout [40] is a topic model that uses the links between previous bug reports and their relevant buggy files to predict the location to be fixed for new bugs. The results show that BugScout outperforms the original LDA model as an LDA variant that incorporates historical bug reports.

Davies et al. [3] train a binary decision tree classifier for every method in the source code. Each transaction represents a source code method; the features are terms, and the class labels indicate whether or not a document is related to the method. The classifiers use the term similarity between the bug reports to predict whether a new report is related to a particular method.

BugLocator [7] is a bug localization approach that uses similar report components. Several studies [5], [32], [43] have used BugLocator as their similar reports component in order to locate bugs more effectively. The similar reports component of BugLocator is introduced in detail in Section 4.3.7, since it is used in experiments in the current study.

BLIA [50] combined structured text information (BugLocator [7]), code change history (AmaLgam [5]), stack traces (BRTRacer [32]) and similar reports (BugLocator [7]), and gave better performance than each approach used individually. The granularity of BLIA was improved from file level to method level by extending the bug repository data. The linear weights were set to fixed values in the range [0.1] to demonstrate its performance in testing.

2.3.7. Execution

Although some report-oriented bug localization approaches use execution traces to improve performance, our goal is to build an automatic framework for bug localization in which after the bug report is input, the system provides the ranked entity list or the predicted buggy entities directly. Execution traces are generally derived from running pass/fail test cases. This dynamic approach requires more work than simply providing

the bug report and other attributes that can be calculated or derived from the source code repository, and we therefore do not consider this attribute in the current paper.

SITIR (Single Trace and Information Retrieval) [11] combined the IR technique of LSI and a dynamic method involving a single-scenario execution trace. The main idea is to filter out methods that were properly executed using predefined scenarios, based on the report description from the ranked methods list derived from LSI algorithms. PROMESIR [12] is a hybrid method that combines LSI with scenario-based probabilistic ranking (SPR) [58]; the latter is a dynamic method that determines the relationship between features and entities depending on the trace of execution.

Adaptive multi-modal bug localization (AML) [46] integrates spectrum-based bug localization, *tf-idf* cosine similarity, and a suspicious word score. The main idea behind the suspicious word component is that words occurring in failing execution traces are more suspicious than other words; methods containing suspicious words tend to be buggy. Dao et al. [59] studied the influence of execution information (i.e. the coverage, slicing and spectrum) on IR-based bug localization. They concluded that both coverage and slicing information can improve IR-based techniques at both class and method levels, but that spectrum information can only improve localization at the method level rather than the class level.

2.3.8. Discussion

In the hybrid bug localization approaches introduced above, there is a trend whereby the performance can be increased by combining suitable beneficial features. For example, the authors of AmaLgam [5] added a version history component to an approach based on BLUiR [4] and BugLocator [7]; AmaLgam then outperformed both of these approaches. Two further components (stack trace and reporter information) were added to AmaLgam+ [48] to improve AmaLgam. Ye et al. [49] added structured textual similarity through BLUiR and web link scores to the approach used in the prior version [45] to improve the localization performance. BLIA [50] combined four components from three other approaches, and outperformed these three approaches used individually. Thus, finding and combining more suitable beneficial features together is a way of improving bug localization performance. However, a combination of more features does not always perform better [59], and a choice of suitable features and combining methods is therefore important.

In addition to the seven main types of feature discussed above, others are used in bug localization, such as the reporter information component in AmaLgam+ [48], the API documents of the source code [45], [49], and so on, although these features are not as widely used. Further features that are more beneficial are likely to be found and the taxonomy of feature types will also be extended in the future.

2.4. Other Related Work

Several literature and questionnaire surveys are introduced in this section.

Dit et al. [1] presented a systematic literature survey of feature location techniques including 89 articles from 25 publications, appearing between November 1992 and February 2011. The investigated approaches can be classified into three main types and combinations of these: dynamic, static and textual methods. Dynamic methods involve the running of test cases, while static methods are based on analyzing the source code structure and dependence, without using execution information. Textual methods use textual information about the features and software. Rubin and Chechik [60] present a survey of 24 feature location approaches, 22 of which were discussed in [1], and provide more details for all of these, especially in terms of the use of call graph dependencies. They classify feature location methods into two types, dynamic and static methods, which depend on whether execution information is used during the running of the software [60].

Wang et al. [61] report an exploratory study of the feature location process without the use of specialized tools. They summarize the feature location process into three easily understood levels: phases, patterns, and actions. This conceptual framework can improve the performance of inexperienced developers in feature location tasks when they have access to conceptual knowledge. The study also shows that human factors, task properties, and in-process feedback are three main external factors influencing the feature location process. Bettenburg et al. [62] conducted a survey of the developers and bug reporters of three projects to determine what makes a good bug report. The results showed that the steps to reproduce and stack traces are most useful in bug reports. Severe problems involve errors in the steps to reproduce, incomplete information, and incorrect observed behavior. Bug duplicates are not considered by developers to be harmful. The authors also asked developers to rate the quality of bug reports, and developed a tool

called CueZilla that automatically predicts the quality of bug reports based on whether the report contains useful elements (stack trace, etc.) and readability.

Herzig et al. [63] manually examined more than 7,000 issue reports from the bug repositories of five projects, and found that 33.8% of bug reports were misclassified. Since bug localization approaches typically use a dataset from the project bug repository [26], these authors strongly recommend that bug data should be manually examined in related studies, in order to reduce the impact of misclassified bugs.

2.5. Summary

In this chapter, we have reviewed the background of IR-based feature location and bug localization. A simple introduction to IR was presented in Section 2.1, and the background and main process of IR-based bug localization are introduced in Section 2.2. In Section 2.3, we introduced recent hybrid methods, which is background knowledge for the approach presented in Chapter 4. Other related work is presented in Section 2.4. Further related work is introduced in the following chapters.

Chapter 3. A Strategy to Determine When to Stop Using Automatic Bug Localization

3.1. Introduction

Bugs are unavoidably generated during software development, and a crucial task during software maintenance is fixing bugs submitted by users. The location of these bugs is a primary task for the software developer, and is tedious and expensive. As such, bug localization activities that aim to find the relevant source code entities (methods, classes, etc.) that need to be modified in order to fix the bug, based on the bug report [1], have become a crucial part of software development. Developers usually name classes, methods, variables and other parts of the source code for their functionalities, and write comments that include terms relevant to the functionalities of source code entities; likewise, bug reports usually include terms relating to buggy functionalities. Thus, bug reports are likely to contain terms that are textually similar to the relevant source code entities [4].

IR-based bug localization resembles a web search engine such as Google or Yahoo. The bug report forms a text query, and the source code entities resemble the webpages that are retrieved. A web search engine outputs a ranked list of webpages to a user when a text query is input, and in the same way, bug localization provides a ranked list of source code entities to developers based on the textual bug report. The higher the rank of the entity in the final list, the more likely it is that the entity is buggy. Developers are therefore tasked with checking the entity list from the beginning until the first relevant (buggy) entity is found.

However, a problem with a technique based on ranked lists is that buggy entities ranked at the bottom of the results list will take a great deal of time to identify, since higher ranked entities are examined before this buggy entity is reached. This is not a wise or realistic method to employ in view of the scarce testing resources available to software developers during debugging. Developers tend to discontinue using IR-based techniques if the relevant entities are not ranked highly enough [64]. Conventional bug localization

approaches in which automatic methods are not used to find bugs are known to be more efficient than using a ranked list when the rank of the relevant entity is low. We assume that using the conventional approach involves a fixed time cost for a specific bug. Previous automatic techniques [4], [15], [32], [35], [39], [42], [43] generally do not consider the situation of using conventional localization approaches, while in reality, developers have the option to use the conventional approach besides only using the automatic localization result.

To address the issue of combining the conventional bug localization approach into automatic techniques, we assume a bug localization process in which after a source code entity in the ranked list has been examined, the developers have the option to continue examining the automatic bug localization results or to change to using the conventional localization approach [6]. We therefore propose a new evaluation metric called ETC which indicates the total expectation for the time cost, including for the conventional localization approaches. The value of this metric is in direct proportion to the time cost expected in the real world. The simulated process of bug localization aims to minimize the value of the ETC for each bug. By using the ETC as the evaluation metric for different localization techniques, developers can determine approximately how much time it will take for a bug to be localized.

The widely used IR-based localization techniques proposed in previous studies are mainly comprised of information retrieval metrics such as Top-N (the number of bug reports in which at least one of the relevant entities appears in the top N results in the ranked list), MAP [18] and MRR [28]. The values of these three evaluation metrics are all based on the rank of the relevant entities in the list, meaning that they only calculate the time spent by developers in examining entities ranked more highly than the first buggy entity. These evaluation metrics and previous localization techniques do not consider a situation in which developers switch to the use of conventional localization approaches when an automatic localization technique is not effective for a bug report, in order to find the specific location of the buggy file. In other words, when conventional approaches are not considered in the bug localization process, IR-based evaluation metrics cannot reflect the true performance of the entire localization process.

The contributions of this section are as follows:

1. We create a hybrid bug localization simulation process that includes the use of the conventional approach. To the best of our knowledge, previous automatic IR-based bug localization techniques do not take conventional approaches into account.
2. We propose a new evaluation metric for the simulated bug localization process called ETC, which is the expected time cost of localizing a bug, including the time expended on using the conventional approach to localize the bug. Minimizing ETC is the goal of the localization process.
3. We propose an optimizing strategy and derive simple criteria to minimize the ETC. These can be used to suggest to developers that they should stop using the automatic bug localization results and switch to the conventional approach.
4. We compare the performance of automatic bug localization technique called BugLocator [7], both with and without our strategy. The results show that the strategy is very effective in comparison to the sole use of automatic localization results or the sole use of conventional approaches, in two real projects.

The rest of this chapter is structured as follows. In a simulated process of bug localization, we make a series of assumptions, as introduced in Section 3.2, including for example that the time cost of using conventional approaches to localize a bug is set to a fixed value for each bug report, and that when developers switch to using conventional approaches, they do not return to using the ranked list. The probability of a source code entity being buggy is discussed in Section 3.3. We derive simple criteria for when to stop using the ranked list provided by the automatic localization technique and to switch to conventional technology, based on the calculation process for ETC that is introduced in Section 3.4. Although the model assumptions and derivational process are sophisticated, the final criteria derived are relatively simple; developers can refer to these criteria and follow the indications for when to discontinue IR techniques in order to minimize the expected localization time cost. A discussion of this approach and the relevant threats to its validity are introduced in Sections 3.5 and 3.6. A conclusion is presented in Section 3.7.

3.2. Assumptions

When automatic bug localization techniques are applied to a given bug report, a list of ranked source code entities is presented to the developers for manual inspection. Wang

et al. [64] found that when using this ranked list, developers can locate faulty files efficiently when the bug report contains rich information; however, the ranked list is less helpful when the bug report contains insufficient information or when its style is inappropriate for the implemented localization approaches. Hence, there are some circumstances in which source code entities related to the bug report are ranked too low to be easily identified and used. Since the identification time required by conventional localization approaches tends to be stable, this means that in some instances in which automatic bug localization techniques are used, developers actually spend more time on identifying the entities in the ranked list than in the conventional approach. Thus, a better strategy would be to choose the conventional localization approach when the bug report is too hard to localize using automatic techniques. We propose a strategy that enables developers to decide when to stop using the ranked list produced by automatic techniques and to switch to the conventional localization approach.

3.2.1. Common Assumptions

To create a model simulating the behaviors involved in localizing bugs in the real world, we first make the following assumptions. The first four assumptions are widely used in bug localization, and we therefore refer to these as common assumptions.

***Assumption 1.** The goal of bug localization is to find the one most relevant source code entity. We assume the developers can easily find other buggy entities after checking the first one found.*

Source code entities refer to the program elements such as classes, files, methods etc. [25]. Recent IR-based bug localization methods generally use *files* as their level of granularity, meaning that these techniques output a ranked list of source code files [4], [5], [7], [39]. We use the word *entity* to refer to the file or method which needs to be localized, so that the description can be used at a various levels of granularity.

The Top-N and MRR [28] evaluation metrics are based on this assumption. Both metrics only consider the highest ranked source code entities, and are therefore concerned with the first buggy entity in the ranked list, ignoring low-ranking buggy entities.

In this study, we focus on IR-based bug localization since spectrum-based techniques [33], [34], [65] typically provide results at statement or instrument level, and this is not consistent with our assumption; developers cannot determine whether or not a source code instrument is buggy by examining only a line of code. However, the method or file

levels are different, since a method or a file provides a large amount of information to developers, thus aiding in determining whether or not it is buggy.

***Assumption 2. Atomicity:** The developer reads the given entity from beginning to end, and then determines whether the entity is buggy. The developer will not pay attention to other entities until the entire content of the given source code entity has been checked.*

This assumption aims to guarantee that each entity is an individual unit, and that entities are processed one by one. Unlike the LOC evaluation metric, other metrics that widely used in evaluating bug localization techniques such as Top-N, MAP and MRR all follow this assumption.

***Assumption 3.** Each source code entity within a given project incurs the same time cost for an inspection by a developer to determine whether or not it is buggy. The time cost of checking a single entity can be set as T for the whole project.*

Top-N, MAP and MRR are all based on this assumption, and treat a source code entity as a single unit. Regardless of the size of an entity, it can be considered to incur the same cost in terms of time for a developer to inspect. Although the lengths of source code entities vary in a real software development scenario, they are treated as being the same length in this evaluation, for simplicity.

***Assumption 4. Ideal developers:** Developers can precisely determine whether or not a source code entity is buggy after checking all the contents of the entity.*

This assumption guarantees that developers will not miss buggy entities when checking them, and also will not wrongly label clean source code entities as buggy.

All of the above four assumptions, or parts of them, are widely used in prior bug localization studies. Although these studies may not directly address these assumptions, they are considered to hold, as a default rule. These assumptions are explained in detail here to enable a better understanding of our proposed strategy.

3.2.2. Specific Assumptions

The following assumptions form the main basis of our model, and are proposed for the first time in the domain of bug localization. We refer to these as specific assumptions in our strategy. The last assumption is introduced in Section 3.3.1.

Assumption 5. *All developers have their own approach to localization in order to find a buggy file without the use of automatic techniques, and this is referred to as a conventional bug localization approach. This approach includes an examination of the stack trace in bug reports, checking the source code based on a description of the bug, reproducing the bug in execution, and so on. We assume that a developer first checks the source code entities in the ranked list, starting from the top, and may later discontinue using the ranked result in the checking process if no buggy entities are found, returning to a conventional approach in order to localize the bug. The developer will not switch back to using the provided ranked list to localize the bug, once a conventional approach has been chosen.*

This assumption is used to simplify the behavior of bug localization. In reality, developers may switch back and forth between the entities on the ranked list and other entities using a conventional approach [64]. We assume that when developers discontinue using the ranked list, this means that they have lost confidence in the automatically produced result, and therefore will not later return to using this again.

Assumption 6. *The time cost of using a conventional localizing approach to find the first buggy entity can be set to T_b for a bug report b .*

It is hard to define the time cost of bug localization in conventional approaches. We set this to a constant value T_b to represent the cost to the developer that can be expected for each bug on an empirical basis. Different bugs have different expected times for localization and fixing. T_b also varies for different developers.

Assumption 7. *There is no intersection between the manually checked entities in the ranked list and a conventional localization checking route.*

This assumption is made in order to simplify the model, and to ensure that the value of T_b will not be influenced by entities that have been checked using the ranked list. Since the entities checked from the ranked list and the entities checked using the conventional localization technique only occupy a small proportion of the total source code, there is little chance of the two types of entities intersecting.

3.3. Probability of Being Buggy

In our strategy for bug localization, the probability of the entity being buggy in the process of checking the ranked list is very important. This probability and the time required to localize the bug using conventional approaches, T_b , influences the decision as to when to stop using the ranked list. More details will be given in the next section. In this section, we first introduce the definition of the probability that the first entity in the ranking list will be buggy. Following this, we use real data to illustrate the monotonically decreasing nature of this probability.

3.3.1. Definition

Definition: The probability of the rank k^{th} entity of the result list being buggy in the examining process is defined as P_k . Since the localization task involves finding the first buggy entity, when we calculate the probability P_k , the assumption is made that the entities with rank higher than k have been checked, and that the developers did not find any bugs.

The probability of the rank k^{th} entity not being buggy in the examination process is defined as P'_k :

$$P'_k = 1 - P_k \quad (3.1)$$

We use Eq. (3.1) to simplify the formulae in the bug localization simulation process. Since there is no guarantee that buggy entities will or will not be found at rank k , we have the following assumption:

$$\forall k, 0 < P_k < 1, \quad \text{and } 0 < P'_k < 1 \quad (3.2)$$

The localization result in Section 3.3.2 shows that, in general, P_k at a higher rank is larger than that at a lower rank. In particular, P_1 has the largest value, meaning that the top source code entity in the final list has the largest probability to be buggy. So we have the following assumption.

Assumption 8. Monotonicity: The probability of entities with rank k being found to be buggy in the examining process, P_k , monotonically decreases, and $\forall k, 0 < P_k < 1$, and $0 < P'_k < 1$.

This monotonicity will be proven using real data, as shown in the next subsection. This assumption also forms an important part of the strategy derivation.

3.3.2. Real Data

The BugLocator dataset [7] has been used in several bug localization approaches [4], [5], [32], [43], and the authors have provided both tools and data. We analyzed the results of the BugLocator² algorithm for the SWT³ and Eclipse⁴ projects, and counted the numbers of buggy entities with each rank in the list of results. For each bug report, we counted only the entity with the highest rank, since developers need only to find the first buggy entity (Assumption 1). The numbers of buggy entities for each rank in the results list are shown in Figure 3.1 and Figure 3.2 for the two projects. To avoid an undefined number (logarithm of zero) on the logarithm scale of the figures, all the numbers of buggy entities have been incremented by one. The figures record the counts of buggy source code entities for all bug reports for each project.

Table 3.1 Number of buggy entities in the top 10 of the ranked lists of all bugs for each project

Rank	#Bugs-Eclipse	#Bugs-SWT
1	981	35
2	319	13
3	172	8
4	132	6
5	94	6
6	75	2
7	64	2
8	52	2
9	53	3
10	42	1

Table 3.1 shows the numbers of buggy entities in the top 10 of the ranked list. Table 3.1 and Figure 3.1 and Figure 3.2 show the same figures in different formats. For example, the first cell in Table 3.1, which has the value 981, indicates that 981 buggy entities can

² <https://code.google.com/p/bugcenter/>

³ <http://www.eclipse.org/swt/>

⁴ <http://www.eclipse.org/>

be found by examining only the first entity in the ranked list generated by the BugLocator tool. The point (1, 982) (981 incremented by one to avoid a value of $\log(0)$) in the top left-hand corner of Figure 3.1 represents the same number.

From Table 3.1, Figure 3.1, and Figure 3.2, we can see that in both projects, the numbers of buggy entities monotonically decrease with the rank of the entities in the list of results. This supports Assumption 8, since the count of buggy entities can be easily transformed to the probability of being buggy, as explained in the following subsection.

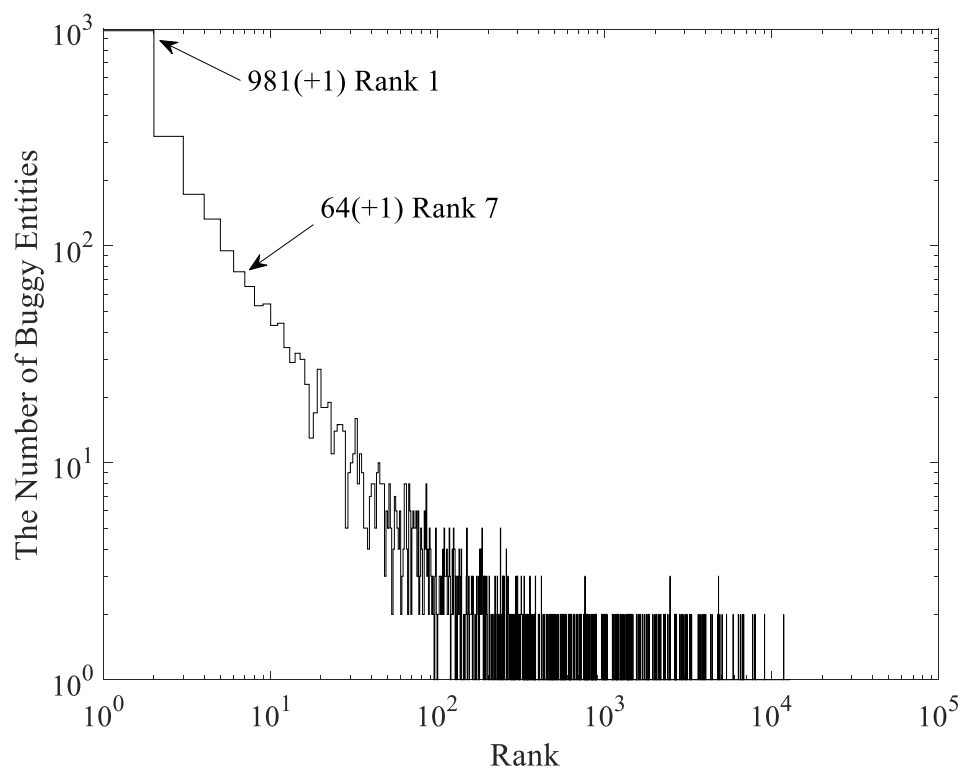


Figure 3.1 Number of buggy entities in the ranked list: Eclipse
(Value incremented by one to avoid $\log(0)$)

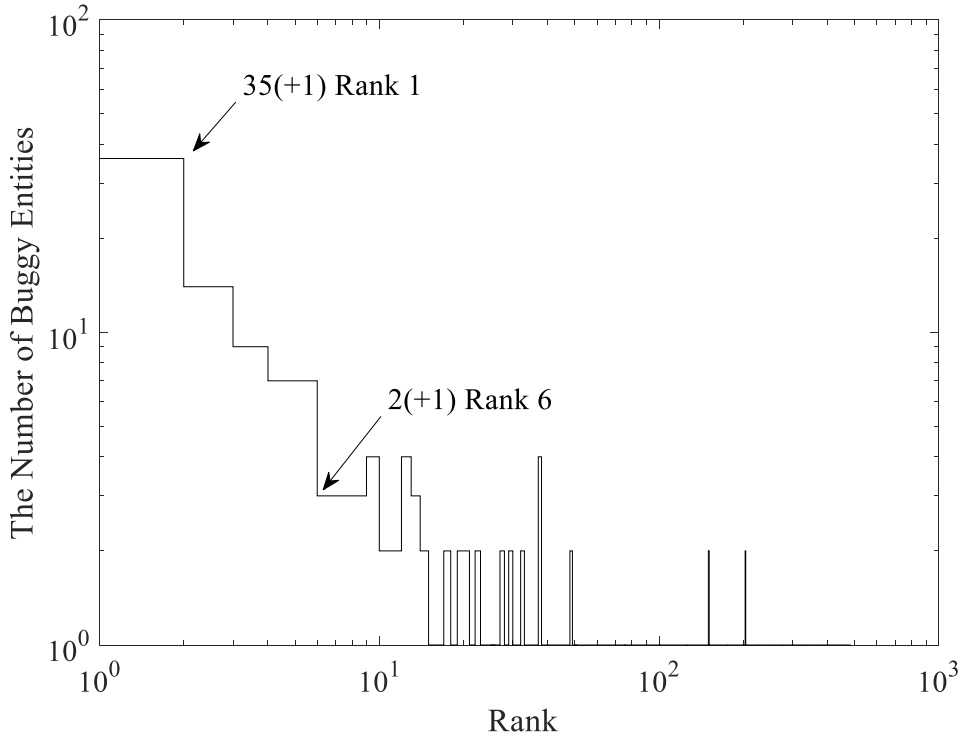


Figure 3.2 Number of buggy entities in the ranked list: SWT
(Value incremented by one to avoid log(0))

3.3.3. Classical Probability Model

A simple way to use localized bug data to calculate the probability P_k is by using the classical probability model, as follows:

$$P_k = \begin{cases} \frac{N_1}{N} & , \quad k = 1 \\ \frac{N_k}{N - \sum_{i=1}^{k-1} N_i} & , \quad k > 1 \end{cases} \quad (3.3)$$

where N is the total number of buggy entities, and N_i is the number of buggy entities with rank i in the resulting list of all bugs. $(N - \sum_{i=1}^{k-1} N_i)$ is the number of candidate entities that are buggy, since the previous $\sum_{i=1}^{k-1} N_i$ entities have been examined and labeled as buggy entities with higher rank. According to Table 3.1 Number of buggy entities in the top 10 of the ranked lists of all bugs for each project, $N_I=981$ and $N=3075$ for Eclipse, and $N_I=35$ and $N=98$ for SWT.

Table 3.2 Probability P_k for the top 10 entities in the ranked list

Rank k	P_k : Eclipse (%)	P_k : SWT (%)
1	31.9	35.7
2	15.2	20.6
3	9.69	16.0
4	8.23	14.3
5	6.39	16.7
6	5.44	6.67
7	4.92	7.14
8	4.20	7.69
9	4.47	12.5
10	3.71	4.76

This constitutes a rough deviation method for P_k . Particularly when k is large, the numbers of buggy entities at low rank, N_k , approach zero, as shown in Figure 3.1 and Figure 3.2, and it is therefore unrealistic to use this method to derive P_k at very low ranks. However, in our strategy, we need only calculate P_k in the top ranks. The calculations of P_k values for both projects using Eq. (3.3) are shown in Figure 3.3 and Figure 3.4, respectively; the values of P_k for the top 10 ranks are shown in Table 3.2. All values of P_k in these two figures are incremented by 0.001 to avoid a $\log(0)$ value. This procedure does not affect the original form of the curve, but makes the figure clearer.

Figure 3.3 and Figure 3.4 also provide data that support Assumption 8. According to Eq. (3.3), when $N_k = 0$, $P_k = 0$. The value of P_k of 0.001 (which has been incremented by 0.001 to avoid $\log(0)$) in the two figures indicates that the corresponding number of buggy entities N_k is zero. We also know that the last $P_k = 1$, since there is only one candidate entity to choose from, as shown in the far right-hand peak in the figures. When the rank is high, it is obvious that P_k monotonically decreases with k ; when the rank is low, since the denominator of Eq. (3.3) is decreasing, if the numerator of Eq. (3.3) $N_i > 0$, P_i will be quite large based on Eq. (3.3). In fact, since most of the numerators of Eq. (3.3), N_i , for low ranks are zero, the use of a smoothing technique can make the decreasing curve clearer. Moreover, we ignore the low ranks in our strategy, since the critical condition of the strategy does not occur at low ranks (see Section 3.4.2).

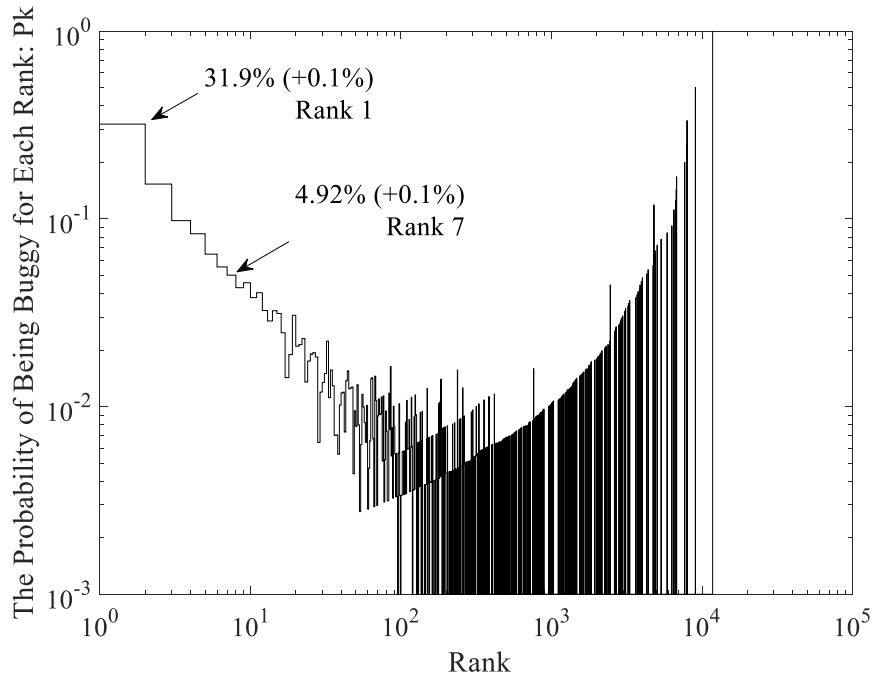


Figure 3.3 P_k for each rank: Eclipse
(value incremented by 0.1% to avoid $\log(0)$)

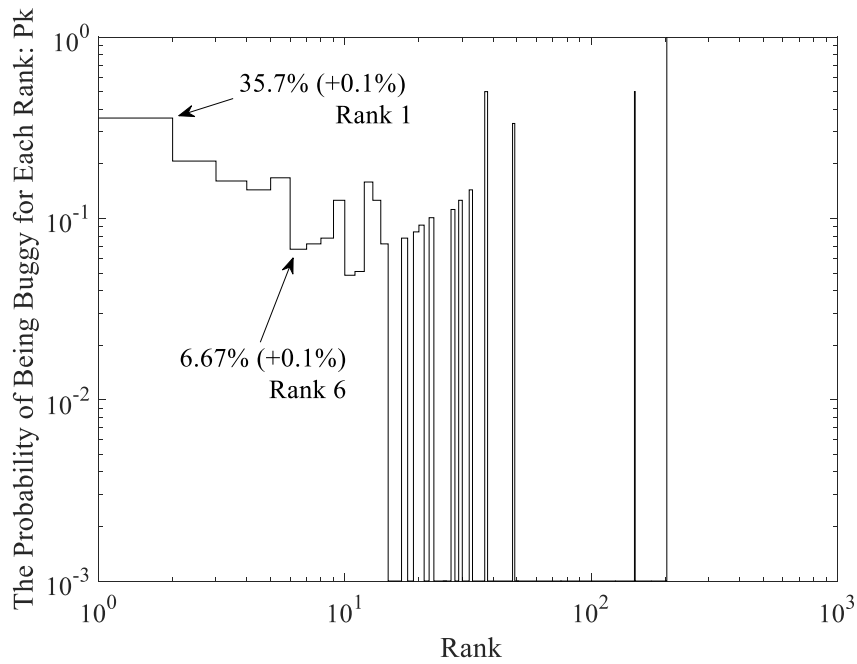


Figure 3.4 P_k for each rank: SWT
(value incremented by 0.001 to avoid $\log(0)$)

3.4. Simulation of the Localization Process

In this section, we simulate the process of bug localization, including the conventional localization approach, and derive a criterion for when to stop using the list of results from automatic localization.

3.4.1. Process Simulation

The cost of checking a source code entity with rank k in the list is defined as $Cost_{Find}(k; X)$, where the meaning of X is defined as follows. In the bug localization process for bug b , when X entities from the ranked list have been examined and no buggy entities have been found, the developer decides to stop using the ranked list and to switch to a conventional approach. The goal of this model is to optimize the value of X . If X is too small, the developer may switch to a conventional approach too early for efficient use of the ranked list. If the value of X is too large, the developer may spend too much time checking the ranked list when this bug localization technique is not effective for this report.

The time cost expectation when the first buggy entity is found at rank k of the ranked list is:

$$Cost_{Find}(k; X) = \begin{cases} TP_1, & k = 1 \\ kTP_k \prod_{i=1}^{k-1} P'_i, & 1 < k \leq X \end{cases} \quad (3.4)$$

If $k = 1$, the time cost equals T , and the probability of finding the buggy entity at the first rank is P_1 . If $1 < k \leq X$, the time cost equals kT . Since k entities have been examined, the probability of finding a buggy entity at rank k is $P_k \prod_{i=1}^{k-1} P'_i$. The reason for this is that developers have not found a buggy entity in the top $k-1$ entities, and finally find a buggy one at rank k .

The time cost expectation after the developer has examined X ($X \geq 0$) entities without finding any buggy entities is as follows:

$$Cost_{NotFind}(X) = \begin{cases} T_b, & X = 0 \\ (XT + T_b) \prod_{i=1}^X P'_i, & X > 0 \end{cases} \quad (3.5)$$

A value $X = 0$ indicates that the developer does not use the results from the IR technique and switches directly to a conventional approach; in this case, the time cost is T_b . $(XT + T_b)$ is the time cost of examining X entities and using conventional methods to localize

bugs, and $\prod_{i=1}^X P'_i$ is the probability of not finding any buggy entities in the top X ranked entities.

The time cost expectation for bug report b , for a developer who stops using the ranked list after checking X entities in the list without finding buggy entities, is the sum expectation of finding buggy entities in the top X entities in the ranked list and the conventional localization cost if no buggy entities are found in the top X entities:

$$Cost_{Total}(X) = \begin{cases} Cost_{NotFind}(X), & X = 0 \\ \sum_{k=1}^X Cost_{Find}(k; X) + Cost_{NotFind}(X), & X > 0 \end{cases} \quad (3.6)$$

The average total cost is the evaluation metric, ETC, which we want to minimize. We use $Cost_{Total}(X)$ to indicate the ETC for a specific bug, which can be distinguished from $Cost_{Find}(X)$ and $Cost_{NotFind}(X)$.

After substituting Eqs. (3.1), (3.4) and (3.5) into (3.6), we obtain the following:

For $X = 0$,

$$Cost_{Total}(X) = T_b \quad (3.7)$$

For $X = 1$,

$$\begin{aligned} Cost_{Total}(X) &= TP_1 + (T + T_b)P'_1 \\ &= T + T_bP'_1 \end{aligned} \quad (3.8)$$

For $X > 1$,

$$\begin{aligned} Cost_{Total}(X) &= TP_1 + T \sum_{k=2}^X (kP_k \prod_{i=1}^{k-1} P'_i) + (XT + T_b) \prod_{i=1}^X P'_i \\ &= T \left(1 + \sum_{k=2}^X \prod_{i=1}^{k-1} P'_i \right) + T_b \prod_{i=1}^X P'_i \end{aligned} \quad (3.9)$$

3.4.2. Optimization of ETC

The goal of this model is to determine the optimized value of X , X_{ideal} , which minimizes the total time cost expectation $Cost_{Total}(X)$:

$$X_{ideal} = \arg \min_X Cost_{Total}(X) \quad (3.10)$$

To optimize the total time cost, we use the difference to obtain:

$$\begin{aligned} \Delta Cost_{Total}(X) &= Cost_{Total}(X+1) - Cost_{Total}(X) \\ &= \begin{cases} T - T_b P'_1, & X = 0 \\ (T - T_b P_{X+1}) \prod_{i=1}^X P'_i, & X > 0 \end{cases} \end{aligned} \quad (3.11)$$

We set $\Delta Cost_{Total}(X) = 0$ in (3.11), and obtain a critical value for P_{X+1} with the condition $\forall i, P'_i > 0$ for $X \geq 0$:

$$P_{X+1} = \frac{T}{T_b} \quad (3.12)$$

From Eqs. (3.11) and (3.12), we can derive:

$$\text{If } P_{X+1} < \frac{T}{T_b}, \Delta Cost_{Total}(X) > 0; \quad (3.13)$$

$$\text{If } P_{X+1} > \frac{T}{T_b}, \Delta Cost_{Total}(X) < 0 \quad (3.14)$$

According to Assumption 8, P_{X+1} monotonically decreases in terms of X when the rank is high, so from Eq. (3.11), $\Delta Cost_{Total}(X)$ monotonically increases in terms of X . In the localization results of the previous study, a large proportion of buggy source code entities are typically ranked at the top of the list provided by IR-based approaches, and with increasing X , P_{X+1} decreases and approaches zero.

This means that the inequality $P_1 > T/T_b$ is likely to hold true, and $P_N < T/T_b$ will also be true in the overwhelming majority of cases, where N is the number of all source code entities.

In this case, $\Delta Cost_{Total}(X) < 0$ when $X = 0$. With increasing X , P_X decreases, and $\Delta Cost_{Total}(X)$ becomes larger. When P_X finally becomes smaller than T/T_b , then $\Delta Cost_{Total}(X) > 0$; in this case, $\Delta Cost_{Total}(X-1) < 0$, and $Cost_{Total}(X)$ is the extreme minimum value. The goal of bug localization is to minimize the cost of the localization task. Thus, X_{ideal} is the minimum value of X , that makes $P_{X+1} < T/T_b$ true:

$$X_{ideal} = \min\{X \mid P_{X+1} < \frac{T}{T_b}, X \geq 0, X \in \mathbb{Z}\} \quad (3.15)$$

If $P_l < T/T_b$, then $\forall X, P_X < T/T_b$, since P_X is assumed to be monotonically decreasing. Thus, according to Eq. (3.13), $\forall X, \Delta Cost_{Total}(X) > 0$, and $Cost_{Total}(X)$ is monotonically increasing. Under these conditions, Eq. (3.15) is still true, and $X_{ideal} = 0$. In other words, if $P_l < T/T_b$, the performance of the automatic localization technique is so bad that developers do not need to examine the ranked list, and should use conventional approaches instead.

To estimate the value of X_{ideal} , the probability distribution of P_X should be revealed. An estimation of P_X from historical bug data was introduced in Section 3.3.3. Since the rank X will be high in this strategy (see Section 3.5.1), P_X can directly be derived using Eq. (3.3) for entities with high ranks, without worrying about the unstable value of Eq. (3.3) for those with low ranks.

3.5. Discussion

3.5.1. Comparison of using or not using the Strategy

In this section, we compare the performance of a state-of-the-art technique, BugLocator [7], both with and without the use of the strategy introduced in this chapter. We use the ETC as an evaluation metric. This comparison uses the BugLocator algorithm for the Eclipse and SWT projects introduced in the previous section.

The time cost of the conventional approach in this strategy needs to be estimated by developers. For simplicity, we set the time cost of the conventional approach T_b in this comparison study to $20T$ for all bugs in the Eclipse project. This means that for all bug reports, 20 source code entities must be examined to localize the bug using the conventional approach. According to the critical condition in Eq. (3.12), $P(X+1)=1/20$. This means that when the probability of being buggy of the next entity from the top of the ranked list to be checked falls below $1/20$, it is suggested that developers stop using the ranked list.

We use P_X , calculated from the classical probability model in Eq. (3.3) and as shown in Table 3.2. For the Eclipse project, $X_{ideal} = 6$, since $P(6+1)=4.92\%$ (shown in bold in Table 3.2) is the first time that $P(X+1)$ falls below $1/20$. The number of source code entities examined (time cost) in the real case for different values of X is shown in Figure 3.5 and Figure 3.6. The value of X is the rank at which developers should stop using automatic methods and switch to a conventional approach.

In the real case of Eclipse, X_{ideal} is also equal to six, with an average of 12.17 source entities needing to be examined to localize a bug. If the conventional approach is used alone, then the average ETC is $T_b = 20T$. If we use only the bug localization results, without using our strategy, we will need to examine 141.8 entities on average. Of course, in reality, developers will not examine as many entities as this; however, we use this value as a way of comparing the ETC for bug localization with and without the stop strategy.

In the case of SWT, we set $T_b = 10T$, and the critical condition as $P(X+I) = 1/10$. As shown in Table 3.2, $P(5+1) = 6.67$ (shown in bold) is the first time $P(X+I)$ becomes less than $1/10$. The real number that needs to be examined to fix the bug is shown in Figure 3.6. Both theory and experimental data support a value of $X_{ideal} = 5$. Developers need to examine only 6.01 source code entities on average to localize a bug in SWT. When using solely automatic methods, developers need to examine an average of 10.03 entities to localize a bug. Using the conventional approach alone will require developers to examine 10 entities for each bug.

We can draw the following conclusion from this performance comparison for the two projects: the use of our strategy (ETC=12.17T, 6.01T) can significantly reduce the time cost compared with using only an automatic localization method (141.8T, 10.03T) and using only a conventional approach (20T, 10T).

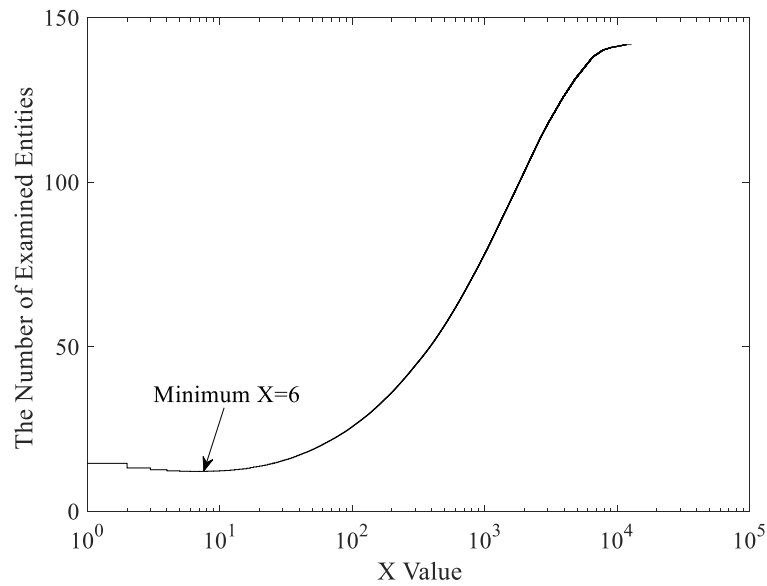


Figure 3.5 Average number of entities that need to be examined to fix a bug for different values of X: Eclipse ($T_b=20T$)

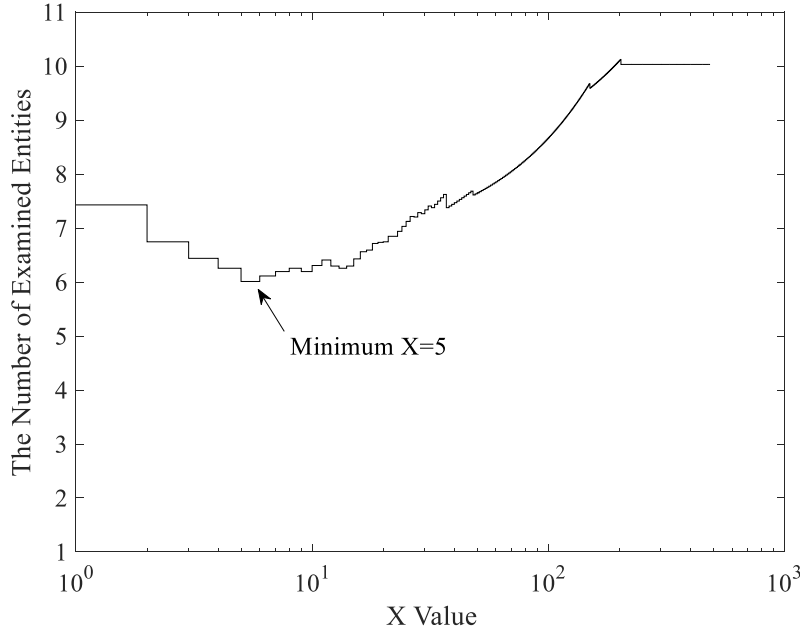


Figure 3.6 Average number of entities needing to be examined to fix a bug for different values of X: SWT ($T_b=10T$)

3.5.2. Time Cost of the Conventional Approach

When using ETC, developers can easily determine how much time is needed to localize a bug; conventional evaluation metrics such as Top-N, MAP, and MRR cannot provide this functionality. In addition, if the average ETC of one localization method is twice that of another method, the former method is estimated to incur a time cost that is twice that of the latter.

Estimating the time cost T_b of a conventional approach is an important part of applying the criterion in real scenarios. In this chapter, we leave the estimation of T_b to the developers. One method for estimating the time cost of bug localization is to use a bug fixing time prediction [66].

Since the probability of finding a buggy entity in the list, P_X , decreases with an increase in rank, and the conventional approach incurs a fixed time cost T_b , a developer does not need to return to using the ranked list after choosing to use a conventional approach.

3.5.3. Another View of the Criterion

Although the final criteria are relatively simple, the derivation is rather complicated. Here, we discuss the meaning of the critical value, T/T_b , from a different perspective. From Assumption 3, we know that T is the time cost for a developer to examine one

source code entity. T_b is the expected time cost using the conventional bug localization approach defined in Assumption 6. We can treat the conventional approach as a process of examining a list of source code entities; the number of examined entities is defined as N_b , where N_b can be a real number (not necessarily an integer):

$$N_b = \frac{T_b}{T} \quad (3.16)$$

Thus, the reciprocal of the critical value T_b/T is the number of source code entities, N_b , that needs to be examined in the conventional approach.

The expected time cost of continuing an examination of the ranked list under the critical condition $P_{X+1}=T/T_b$ in Eq. (3.12) and the expected time cost of using the conventional approach is the same. In the conventional approach, the total time cost for localizing a bug is a constant, $T_b = N_b T$. When continuing to check the $X+1$ entity, the probability of localizing the bug is $P_{X+1}=T/T_b$, as shown in Eq. (3.12).

In this view, the critical value T/T_b can be considered as the probability of choosing a single specific source code entity for examination from all the entities in the conventional list.

3.6. Threats to Validity

There are several potential threats to the validity of our work, as described below.

The critical condition is based on the expected time cost of the conventional approach, T_b . Different developers have different levels of experience, gained in a variety of projects and for a variety of types of bug. Therefore, the conventional localization time cost T_b varies a great deal for different developers, types of bugs and projects, for example. In this chapter, we assume that the value of T_b is estimated by developers based on their own experience. A more precise estimation of T_b will cause the strategy to be more effective. We leave this for future work.

We assume that the examination of each source code entity costs the same time. Previous approaches using Top-N, MAP, and MRR are based on the same assumption. Since the length of a source code entity varies greatly, the time cost for examining an entity also varies. To simplify the derivation of this strategy, we also use this assumption; however, the use of LOC to estimate the time cost may be more precise.

The probability P_X is less stable at high ranks for the larger project, Eclipse, although in the smaller project, SWT, P_X tends to oscillate at high ranks due to the small amount of data. When our strategy is applied to a small project, the calculation of the critical value will be less precise than that for a larger project. For a large project, if there are insufficient historical data on localized bugs, the probability P_X will also be derived less precisely. A suitable smoothing method for P_X may offer a solution to this situation.

3.7. Summary

This chapter introduces a simulated process of bug localization that involves the use of a conventional localization approach if the automatic method is ineffective for a given bug, in contrast to previous automatic IR-based bug localization methods, which do not consider this case. We assume that developers can decide when to stop using the automatic localization results and switch to a conventional approach. In this simulated process, we propose a novel evaluation metric, ETC, to indicate the expectation of the total time cost for localizing a bug; this also includes the time cost of using a conventional localization approach. The goal of determining the point at which the developer should stop using the ranked list requires minimizing the ETC for each bug. The criterion for minimizing the ETC is derived in a relatively simple form, and is expressed in terms of the probability of localizing the bug in the next entity of the ranked list and the expected time cost for localizing the bug using a conventional approach. We compared the time cost of a state-of-the-art automatic localization method called BugLocator, with and without the use of our strategy, for two projects. The results show that the use of our strategy can significantly reduce the time cost compared to using the automatic localization approach alone and using the conventional approach alone.

Chapter 4. Hybrid Bug Localization using Learning to Rank Techniques

4.1. Introduction

A software project in development or maintenance phase most likely contains bugs. Bug localization is the process of using automatic or semi-automatic techniques to find the specific location in source code, which needs to be modified to fix the bugs. It has attracted so much attention in the recent decade. Since the developers usually name the classes/methods/variables after their functionalities and write comments with relevant terms, the submitted bug reports usually include the terms about buggy functionalities. The bug reports are therefore likely to have similar textual terms with its relevant source code entities. Information Retrieval (IR), the main technique employed for bug localization refers to the process of finding documents that satisfy an information need from a large collection of documents. Many IR based bug localization techniques have been studied in recent years [4], [15], [32], [35], [39], [42], [43].

Besides mainly depending on IR similarities, many hybrid approaches using additional features from source code structure [13]–[15], dynamic analysis [10]–[12], version history [5], [9] and others have been proposed in the literature that aims at locating bugs more precisely. These hybrid approaches show significantly better performance when compared to the approaches using only IR similarities. It is because they utilize additional features in favor of bug localization. A survey of these hybrid methods and their corresponding features is conducted in this chapter. However, many features are linearly combined with preset weight values [4], [5], [7], [32], [43], which means the linear weight values are not learned from the data.

Combining the found beneficial features in a suitable way may improve the performance of bug localization. Learning to Rank (LtR), a set of machine learning based ranking algorithms in the information retrieval domain, which has been widely used in web searching area in last ten years [16], provides an automated framework for combining different features in ranking task [67]. Therefore, we chose the LtR techniques to combine these beneficial features in this study.

In this chapter, we compare the performance of eight LtR techniques in bug localization; such an approach combines the features that have been used in other studies. The features are selected from previous hybrid bug localization studies, and the feature weight values in LtR techniques are learned from historical bug data and source code information. When more beneficial features are found in future, they could be imported into this approach to increase the localization precision. The results show that the coordinate ascent algorithm [68], which learns weights in a linear combination of features without normalization, is a suitable means for bug localization in our selected features and projects, and it outperforms two state-of-art localization approaches for two large projects, namely, Eclipse and SWT.

The contributions of this chapter are listed as follows:

1. We present a summary of recent hybrid IR-based bug localization methods and the features used in these hybrid methods.
2. We adopt various LtR techniques in the same framework to locate bugs, combining the features used in the proposed bug localization approach. LtR techniques use part of historical bugs to train the learner and test on current bugs.
3. We compared the performances of 8 LtR techniques with 4 normalization methods for bug localization on real projects. The results show the coordinate ascent algorithm [68], which learns weights in linear combination of features, without normalization is a suitable way for bug localization in our selected features and projects, and it outperforms two state-of-art localization approaches for two large projects Eclipse and SWT.

The background of hybrid bug localization and the survey of hybrid methods has been introduced in Section 2.3. LtR techniques and our LtR based hybrid approach are introduced in Section 4.2. We introduce the features we selected in the LtR based approach and show the results of applying LtR techniques in bug localization in Section 4.3. The dataset and tools used are introduced in Section 4.4 and 4.5. We present the experiment result in Section 4.6. The threats to validity are discussed in Section 4.7. The brief summary is shown in Section 4.8.

4.2. Learning to Rank Based bug Localization

In the hybrid bug localization methods introduced in Section 2.3, features are usually linearly combined with preset weight values [4], [5], [7], [32], [43], but the linear

weights are set to fixed values from 0 to 1 to show the test performance without weight learning process. For example, two feature scores x and y are linearly combined to derive the final score. The final score of file f associated with report r is calculated as follows:

$$FinalScore(f, r) = \alpha x + (1 - \alpha)y, \quad (4.1)$$

where α is configured to vary from 0 to 1, with 0.1 increment. Then, the authors show the bug localization performance with a different α value. If there are three features with two linear weight α and β , α value is first optimized with $\beta=0$ to avoid the impact of third feature, and then β value will be optimized with the fixed optimized α value [5], [50]. We use LtR algorithms to combine various beneficial features that can be learned from historical bug data.

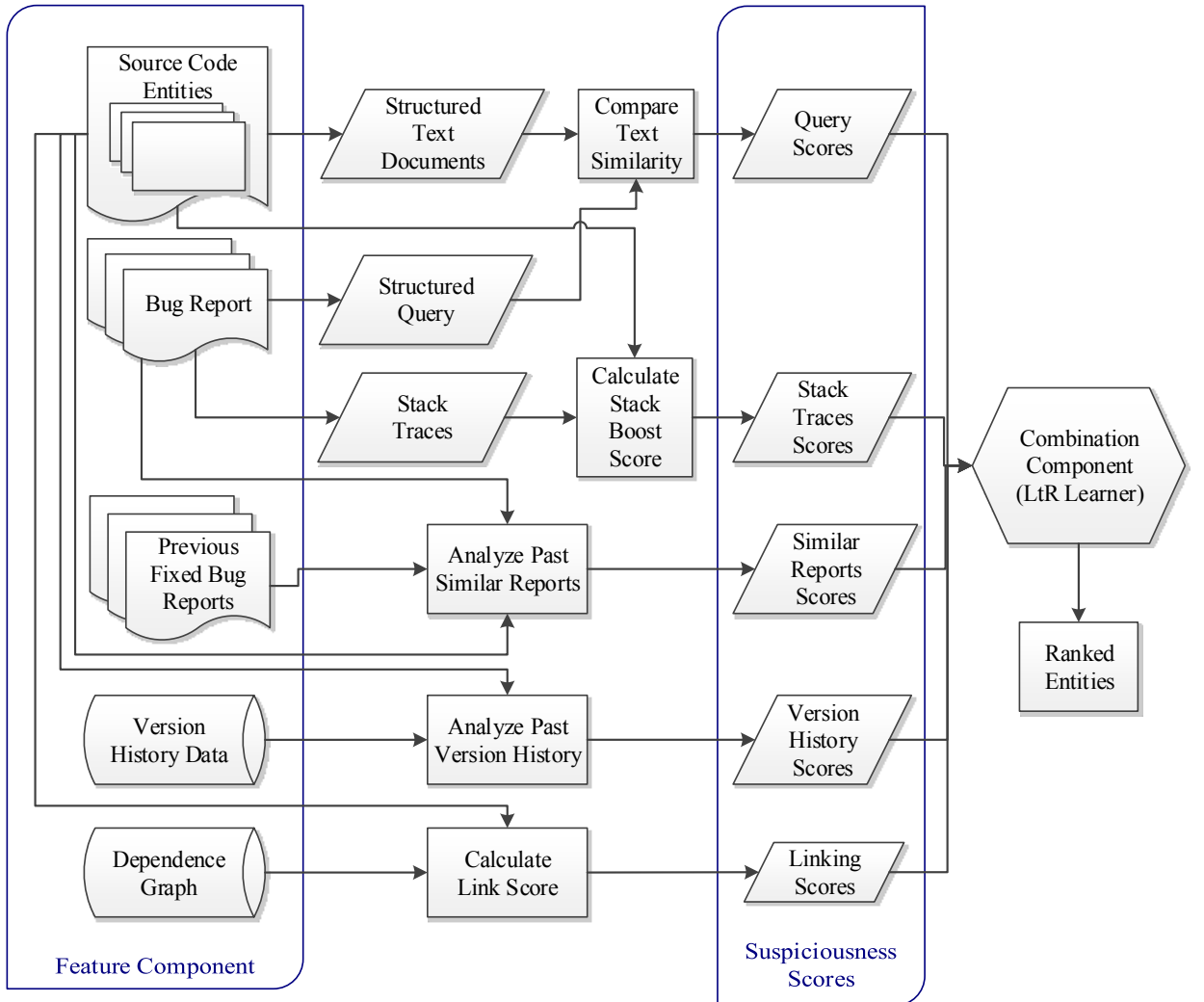


Figure 4.1 The flowchart of the LtR based hybrid bug localization

The flowchart of our planned approach is shown in Figure 4.1. The left part of the flowchart contains selected feature components including the bug report, source code entities and other useful data. The first step is using different techniques to get the suspiciousness scores for each component. The suspiciousness scores indicate the probability of a source code entity being related to a new bug. Each feature component usually has a suspiciousness score, except for the text information and structure of bug reports and source code entities, which will be introduced below. The combination component is used to combine all the suspiciousness scores, and to produce ranked entities result, which is the same as a conventional bug localization technique. When using LtR as the combining techniques, the combination component is a trained LtR learner. In this flowchart (Figure 4.1), we see that all features are related to the source code entities, while version history scores and linking scores are independent of the new bug report. In the experiment, since the dataset provides file-level data, we set the file as the localization granularity.

4.2.1. Learning to Rank

In general, Learning to Rank algorithms are the methods which use machine learning technologies to solve the problem of ranking [16]. Conventional ranking models include Vector Space model (VSM), language model (LM) [18], BM25 [52] and others, and they usually give a similarity score between the query and the document with given parameter values. LtR techniques train learning models on these conventional IR similarities and other features like PageRank [53] from link structure of documents, user feedback and so on, to predict the relevance between the documents and the query in test data.

Ye et al. [49] used the support vector machine (SVM) based ranking method to map bug reports to relevant files. A total of 19 features are selected to learn the SVM model. This SVM ranking method was also compared with two bug localization methods, BLUiR and BugLocator, and showed better performance. Zhao et al. [69] defined three effort-aware metrics, which are all based on lines of code (LOC), to examine the actual performance of LtR bug localization, and claimed that the LtR method is similar or even worse than the standard VSM.

LtR algorithms are grouped into three approaches: the pointwise approach, the pairwise approach, and the listwise approach [16]. The three approaches model the process of

LtR in different ways, and they define different input and output spaces. The details of the three types of approaches are shown in Table 4.1.

Table 4.1 The three group approaches of LtR algorithms

Approach	Input space	Output space	Algorithms
Pointwise	A feature vector of each single document	The relevance degree of each single document	Random Forests [70], MART [71]
Pairwise	A feature vector for each pair of documents	The pairwise preference, which takes values from $\{+1, -1\}$ between each pair of documents	RankNet [72], RankBoost [73], LambdaMART [74]
Listwise	A set of documents associated with query	The ranked list (or permutation) of the documents	ListNet [75], AdaRank [76], Coordinate Ascent [68]

The **pointwise** approach uses the feature vector of a document to predict the relevance score of the document. This approach does not consider the interdependency between documents. If the number of relevant documents for each query varies considerably, the overall prediction will be dominated by queries that have a large number of relevant documents. Given that the ranking problem is more about predicting the relative order rather than the accurate relevance score, the pointwise approach is thus limited.

The **pairwise** approach uses a pair of documents to predict the relative order between them. This approach only considers the relative order between a pair of documents. The position of the documents in the final ranked list can hardly be derived when the focus is only on a pair of documents.

The **listwise** approach takes a set of documents and outputs the ranked list for each query. This approach is more in accordance with the ranking task than the pointwise and pairwise approaches. However, the training complexities of listwise approaches are usually high, and the use of the position information in some listwise ranking algorithms is insufficient [16].

4.3. Experimental Evaluation

4.3.1. Features

The features used in the experiment are introduced in this Section. All the attributes used in the experiment are summarized in Table 4.2. In short, the source code files and the bug report repository including textual report, open date (the time of the bug found), fix date (the time of the bug fixed), and related source code entities are needed for the input part of this LtR approach. We use *component* to indicate the input data type, and *attribute* to indicate the feature values (component scores) of our learner.

Table 4.2 Temporal attributes of input data types

Input data type	Attribute			Description / Formula
Text information with structure: (2 report fields × 5 entity fields)	Bug report summary	×	Class	BM25 based similarities used in BLUiR [4]
			Method	
	Bug report description		Variable	
			Comment	
			Content	
Stack trace	Stack traces score			$BoostScore(x) = \begin{cases} \frac{1}{rank} & x \in D \cap rank \leq 10 \\ 0.1 & x \in D \cap rank > 10 \\ 0.1 & x \in C \\ 0 & otherwise \end{cases}$
Version history	Version history score			$HistroyScore(f, k, R)$ $= \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1 - (\frac{k - t_c}{k}))}}$
Dependence graph	PageRank score			PageRank [53]
	HITS: Hub score			HITS [54]
	HITS: Authority score			
Similar report	Similar report score			$SimiScore(F_j)$ $= \sum_{All\ S_i\ that\ connected\ to\ F_j} (Similarity(B, S_i)/n_i)$

4.3.2. Text Information

In the experiment, the text similarity between source code entities and bug reports are calculated in the same way as BLUiR [4], which is a BM25 based algorithm. There are two parameters in BM25 to control the scale of both term frequencies and document lengths. Since we use the same dataset as BLUiR does, we just use the same values of the best tuning parameters as BLUiR for simplicity.

4.3.3. Stack Trace

We use the boost score of stack trace in the experiment as Wong et al. [32], which is basically the reciprocal of the file rank in the stack:

$$BoostScore(x) = \begin{cases} \frac{1}{rank} & x \in D \cap rank \leq 10 \\ 0.1 & x \in D \cap rank > 10 \\ 0.1 & x \in C \\ 0 & otherwise \end{cases} \quad (4.2)$$

Here, x indicates the source code file; D is the files mentioned in the stack trace; and C is the closely related files of D . The intuition is that the entity with higher rank in the stack trace gets higher boost score, which means it is more suspicious to be buggy. We extract stack traces from bug report descriptions with Java regular expressions, which is similar to the method used by Bettenburg et al. [77]. Each entity in the stack trace is recognized as follows:

at [METHOD] ([FILENAME].java: [LINE])

4.3.4. Version History

The version history component of our work adopts the same formula as AmaLgam [5]. The suspiciousness score is calculated in the following:

$$HistoryScore(f, k, R) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12 \left(1 - \left(\frac{k - t_c}{k} \right) \right)}} \quad (4.3)$$

In the above equation, c indicates a single commit, f is current testing file, R refers to the set of relevant commits, k is a scale parameter which is set to 15 in the original approach, and t_c is the number of days that have elapsed between a commit c and the input bug report. The output of this algorithm is a set of suspiciousness scores, one for each file.

There is some difference between the feature we used and AmaLgam. AmaLgam extracts version history from the VCS (version control system) of the used projects. For simplicity, our work only treats the bug reports and related source code files in the BugLocator dataset as the source of version history component. This means we have less information on this component when compared to the original work.

4.3.5. Dependence

The dependence component in our approach uses PageRank and HITS (including Hub score & Authority score) scores as three attributes in the classifier and aids classifier to

learn the patterns of dependence graph. Other attributes indicating the dependence structure of source code can also be considered.

4.3.6. Text Structure

This component should be considered with the IR similarity component together. Because the text information component also uses the same approach as BLUiR [4], the similarity score between each type pair are the same as BLUiR. There are totally 8 combinations of field pair similarity in BLUiR: four source code fields (class, method, variable, comment) and two bug reports fields (summary, description). To get all terms existing in the source code, we also add another source code field which is named as *content*. The *content* field includes all text terms in a source code entity, and this field can also be considered as a weight balance since it contains all terms which belong to other four source code fields (class, method, variable, comment). The difference between this approach and BLUiR is, we use LtR techniques to combine these text similarity scores and other attributes, instead of simply summing these scores. Unlike other non-textual attributes in the framework, the essence of text structure component is the similarity scores from different text type pairs.

4.3.7. Similar Report

In BugLocator, when a new report B arrives, the textual similarities between B and all pervious fixed reports S are calculated. Then the similar report component score of the file F_j is derived as the following equation:

$$SimiScore(F_j) = \sum_{All\ S_i\ connected\ to\ F_j} \frac{Simialrity(B, S_i)}{n_i} \quad (4.4)$$

All S_i that are connected to F_j means all bugs that were fixed through modifying the source code file F_j , n_i is the number of files that are modified to fix the bug S_i . Other similar report components in previous bug localization approaches are not so popularly used as the BugLocator. So we opt to use BugLocator as our similar report component.

In order to have a unified similarity measure, the similarity measure used in this component will employ the same measure in the text information component, which is based on BLUiR [4]. The difference is that we consider the bug title and description as the two fields in the text similarity, and use the sum of each field pair similarity as the report similarity between the new report B and previous report S .

4.4. Dataset

Table 4.3 Details of BugLocator dataset

Project	Description	Period	#Bugs	#Files
SWT-3.1	Widget toolkit for Java	Oct 2004-Apr 2010	98	484
ZXing	Barcode image processing for Android	Mar 2010-Sep 2010	20	391
Eclipse-3.1	Popular IDE for java	Oct 2004-Mar 2011	3075	11854

There have been several hybrid approaches [4], [5], [7], [32], [43] that used the BugLocator [7] dataset⁵. We can thus compare the performance of our approach to other hybrid approaches conveniently. There are few comparisons between other hybrid bug localization approaches, and those approaches usually use self-extracted dataset. The BugLocator dataset is in file-level. The details are shown in Table 4.3. Although the periods of the three projects are quite long, and there are many versions across these periods, we can only select the earliest version of each project for simplicity as the BugLocator did originally.

4.5. Tools

We use the RankLib⁶ tool, which implements 8 learning to rank algorithms, including MART (Multiple Additive Regression Trees, a.k.a. Gradient boosted regression tree) [71], RankNet [72], RankBoost [73], AdaRank [76], Coordinate Ascent [68], LambdaMART [74], ListNet [75], Random Forests [70], and L2 linear regression.

Although most of the features used in this study have been proposed, we need to develop our own tool to derive the attributes. Because the attributes are intermediate results which cannot be derived from the tools or results which the authors of hybrid methods provided.

JRipples [78] is an Eclipse plug-in that supports feature location. We modify the JRipples tool to extract data including term frequency of different fields in source code and dependence graph, using JDT (Eclipse Java development tools)⁷. We also use the R-

⁵ The dataset website: <https://code.google.com/p/bugcenter/>

⁶ <http://sourceforge.net/p/lemur/wiki/RankLib/>

⁷ <http://www.eclipse.org/jdt/>

project⁸ to preprocess and produce each feature of the approach, including PageRank and HITS scores.

4.6. Result

4.6.1. Comparison to Reproduced Work

The LtR approach uses the same features of related studies, so we can reproduce the BLUiR [4] and AmaLgam [5] results based on our extracted features. AmaLgam uses a linear combination of text similarity, similar report and version history as the final ranking score of a source code entity. The parameters of our reproduced AmaLgam have been tuned to get the best performance as introduced in the original paper [5]. The BLUiR approach uses the sum of 8 features as the suspicious score of a source code file being relevant to a bug report, and the 8 features are 8 combinations of BM25 based similarities (2 bug report fields combined with 4 source code fields). The AmaLgam approach uses the same 8 text similarity features as BLUiR, with additional 2 features including version history and stack traces. Our LtR approaches use the same 10 features as AmaLgam with additional 6 features as shown in Table 4.4.

Table 4.4 The features used in BLUiR, AmaLgam and our LtR approach

Feature(s)		BLUiR	AmaLgam	LtR
8 field combination of text similarity: source code (class, method, variable, comment) * bug report (summary, description)		Yes	Yes	Yes
Similar report		No	Yes	Yes
Version history		No	Yes	Yes
2 field combination of text similarity: source code (content) * bug report (summary, description)		No	No	Yes
Stack trace		No	No	Yes
Dependence graph	PageRank	No	No	Yes
	HITS: Hub	No	No	Yes
	HITS: Authority	No	No	Yes

We developed programs to extract the features which have been proposed in other approaches including BLUiR and AmaLgam. However, there are some differences

⁸ <http://www.r-project.org/>

between our development and the original authors. For example, in text preprocessing and extracting, a little modification in some steps will cause some difference. The performance of our reproduced BLUiR and AmaLgam are different from the ones claimed by the original authors. So we did not use the original evaluation result of BLUiR and AmaLgam. But because our LtR approach is based on the same features of reproduced techniques, it is fair to compare the performance of our LtR approach and the reproduced techniques.

4.6.2. SWT and ZXing Result

In the RankLib parameter setting, we use 5-fold cross validation, and test on all normalizing setting including no-normalizing, sum (normalizing each feature by the sum of all its values), zscore (normalizing each feature by its mean/standard deviation) and linear (normalizing each feature by its min/max values). In the 5 chunks of data, 3 chunks are considered as the train data, 1 chunk for validation to avoid overfitting in the train set, and the last chunk is the test data. Recent bug localization approaches usually use MAP, MRR and TopN to evaluate the results. We select the MAP as the evaluation metric on test data and the metric to optimize on the training data, since RankLib evaluation metric does not support MRR and TopN. For the rest of the parameters, we use the default setting for each algorithm.

Table 4.5 Average MAP result of SWT project in 5-fold cross-validation

Ranker \ Normalization	None		Linear		Sum		Zscore	
	Train	Test	Train	Test	Train	Test	Train	Test
1 MART	0.599	0.472	0.644	0.487	0.681	0.507	0.668	0.497
2 RankNet	0.180	0.182	0.552	0.564	0.422	0.455	0.499	0.518
3 RankBoost	0.509	0.504	0.547	0.544	0.500	0.439	0.528	0.512
4 AdaRank	0.520	0.498	0.558	0.531	0.535	0.513	0.545	0.514
5 Coordinate Ascent	0.621	0.602	0.634	0.590	0.614	0.554	0.633	0.594
6 LambdaMART	0.580	0.445	0.635	0.501	0.677	0.458	0.648	0.525
7 ListNet	0.392	0.383	0.224	0.237	0.129	0.148	0.296	0.274
8 Random Forests	0.981	0.527	0.982	0.579	0.980	0.572	0.983	0.542
9 Linear Regression	0.599	0.472	0.644	0.487	0.681	0.507	0.668	0.497

Table 4.6 Average MAP result of ZXing project in 5-fold cross-validation

<div> <div>Normalization</div> <div>Ranker</div> </div>	None		Linear		Sum		Zscore	
	Train	Test	Train	Test	Train	Test	Train	Test
0 MART	0.698	0.279	0.758	0.346	0.696	0.144	0.744	0.301
1 RankNet	0.253	0.298	0.389	0.381	0.182	0.146	0.371	0.347
2 RankBoost	0.477	0.331	0.379	0.230	0.319	0.192	0.413	0.294
3 AdaRank	0.348	0.274	0.370	0.290	0.377	0.343	0.366	0.312
4 Coordinate Ascent	0.521	0.389	0.506	0.339	0.517	0.331	0.535	0.386
6 LambdaMART	0.769	0.173	0.780	0.282	0.601	0.126	0.715	0.113
7 ListNet	0.200	0.301	0.109	0.084	0.168	0.046	0.245	0.241
8 Random Forests	0.977	0.379	0.975	0.351	0.981	0.330	0.978	0.349
9 Linear Regression	0.504	0.231	0.412	0.301	0.461	0.206	0.491	0.265

Table 4.5 and Table 4.6 show the average MAP result of two projects in 5-fold cross-validation. The maximum test MAP value is highlighted by the bold type in each table. The train/test column indicates the performance value on the train/test data. The random forests algorithm has high metric values on the train data, because these algorithms learned a good model for the train data. But they have relative low performance compared to other rankers on the test data due to overfitting. The MAP performance of rankers like RankNet have a large different between different normalization procedure. From 5-fold cross-validation results of two projects, the coordinate ascent algorithm [68] have good performance on the whole, and the results of coordinate ascent do not have very low performance like some other rankers.

4.6.3. Eclipse Result

We considered the Eclipse project, which is large with 3,075 bugs and 11,854 source code files. To save running time, we divide the bugs into 30 folds in chronological order and use a sliding window setting: one fold as the train set and the next fold as the test set. We use the coordinate ascent algorithm without normalization because it has shown good performance in SWT and ZXing projects in general. To get better performance, a parameter tuning process is added in Eclipse testing, although default parameter values are used in SWT and ZXing projects. First, two parameters of coordinate ascent algorithm are tuned using grid search: restarts = (3, 5, 10, 20) and iterations = (10, 25, 50, 100). The grid search values are selected reasonably with the default parameter value in the middle. We trained the algorithm on the first fold, validated it on the second fold,

and tested it on the third fold. The best model on the validation set is selected to test. We found that the algorithm performed best on the third fold of Eclipse using the default parameter (restarts = 5, iterations = 25). Thus, the default parameters are used in the following evaluation steps.

For the Eclipse project, there are numerous irrelevant files for each bug. Therefore, we first used the BLUiR method to rank all files for each bug, and selected the top 200 irrelevant files and all relevant files for each bug as the train set. The results of applying the coordinate ascent algorithm on the Eclipse project is shown in Figure 4.2. The train curve indicates the MAP values evaluated on the train fold itself. The test curve represents the MAP values tested on the next fold of the training fold. We also attempted

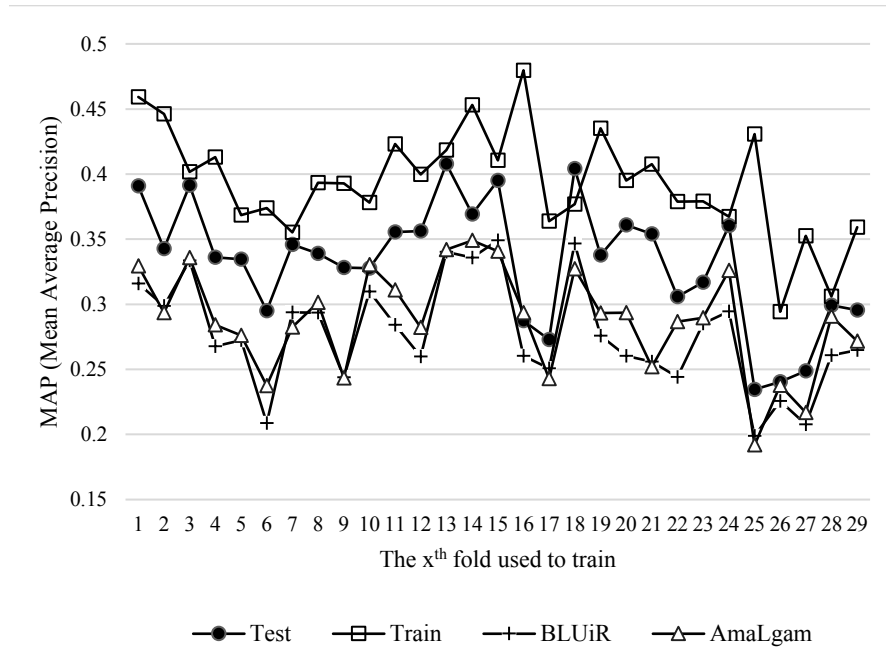


Figure 4.2 The MAP values in folds of Eclipse project

to add 1 fold as a validation set or split the train set for validation. However, one or two folds perform poorly when the validation set is added, and their average MAP values are closed to the ones in the current presented setting (one fold for train and next fold for test). The BLUiR [4] and AmaLgam [5] curves show the performance of reproduced corresponding algorithms in each fold. The BLUiR method does not need to be trained further, and the parameters of AmaLgam is tuned by maximizing the average MAP values of all bugs. We applied Wilcoxon signed-rank test on the MAP values of coordinate ascent, BLUiR, and AmaLgam in each fold. The result shows that coordinate ascent is significantly better than BLUiR ($p = 2.5e-6$) and AmaLgam ($p = 3.9e-6$).

4.6.4. Result Comparison

Table 4.7 MAP comparison of LtR based approach and reproduced state-of-art techniques

Project	Evaluation metric	LtR (coordinate ascent with no-normalizing)	BLUiR	AmaLgam
ZXing	Top1	6	7	8
	Top5	14	15	16
	Top10	15	17	17
	MAP	0.389	0.412	0.447
	MRR	0.459	0.499	0.532
SWT	Top1	55	44	46
	Top5	93	92	91
	Top10	120	117	120
	MAP	0.602	0.505	0.526
	MRR	0.671	0.576	0.598
Eclipse	Top1	914	828	870
	Top5	2043	1934	1975
	Top10	2615	2554	2616
	MAP	0.306	0.278	0.289
	MRR	0.399	0.374	0.389

Given that the coordinate ascent algorithm with no-normalizing performs best in our previous experiments as shown in Table 4.5 and Table 4.6, we select it for comparison with reproduced BLUiR and AmaLgam approaches in the MAP, MRR, and TopN (N=1, 5, 10) metrics, which is shown in Table 4.7. The evaluation results of LtR are derived from the entire project that combined each fold-test set. Unlike SWT and ZXing projects in which the test data are trained and validated on the other four folds, the Eclipse test data are trained on the previous fold in the bug-fixed time order, and the first fold is trained on the last fold. The results of BLUiR and AmaLgam are directly from the entire project because no training parts are present. The results show that the LtR outperforms BLUiR and AmaLgam on the large projects, Eclipse and SWT, by using coordinate ascent without normalization. The performance of LtR is no better than the other two techniques that consider the ZXing project. However, ZXing has only 20 bugs, with 16 bugs used for training and the remaining 4 bugs used for testing in the 5-fold LtR approaches. The small data size of ZXing could be the main reason why LtR did not exhibit good performance. We found that the linear weights of ZXing differed much

more in each fold than did the other projects, thereby indicating that the results of ZXing easily over fit the train set.

By contrast, the data of the Eclipse project have 3075 bugs and many source code files. Even when the project is divided into 30 folds, more than 100 bugs are found for training in each fold. Figure 4.2 indicates that the LtR test performances in most folds are better than or similar to the other two reproduced techniques.

4.7. Threats to Validity

Our approach faces the following threats to validity:

First, we only use the default parameter setting for each LtR algorithm in RankLib. The performance of LtR algorithms should be different when using different parameter values. The next step of this work is testing the influence of different parameter setting in LtR based bug localization.

Second, the bug repository is extracted from the version history, and the relevant files are labeled using the modification commits with text “bug|fix”. So the labeled relevant files may not be the real files which need to be modified to fix the bug. This threat to validity exists in all bug localization approaches using the automatic extraction method without manual examination [79].

Third, we apply learning to rank algorithms to our selected useful features. A different conclusion may be derived using different features. But instinctively, combining more correlated features in a suitable way could increase the localization performance. This area is left for future work.

4.8. Summary

Recent hybrid bug localization methods proposed have examined and adopted many helpful features from version history, similar bug reports, source code structure, etc. But there is little work researching on how to combine these beneficial features. To use a suitable way to combine these features to locate bugs more effectively, we propose a Learning to Rank based hybrid approach on the features which have been found to be useful for bug localization in this chapter. We compare 9 different LtR techniques with 4 normalization methods in bug localization and two reproduced state-of-art approaches.

The results show the coordinate ascent algorithm without normalization is a suitable LtR method for our selected features, and it outperforms two state-of-art approaches for two large projects Eclipse and SWT.

Chapter 5. BM25 and BM25F based Feature Location Techniques

5.1. Introduction

During the software development and maintenance processes, developers generally need to implement new features, improve existing functionalities and fix bugs. The aim of feature location is to help developers to quickly find the part of source code that needs to be added or modified, to implement the required functionalities or remove undesired functionalities such as bugs.

Feature location is defined as the identification of an initial location in the source code corresponding to a specific functionality. Many feature location techniques use bug reports as features in their case studies, since descriptions of the functionality in the requirements document are typically hard to obtain. Some researchers use the term bug localization rather than feature location when the features of interest are bug reports [4], [5], [9], [25], [35], [39].

The functionalities of feature location include the implementation of new features, improvements to existing functions, and the removal of bugs, in a similar way to the removal of unwanted functionalities [1]. Regardless of the types of data used (i.e. bug reports, patch descriptions, enhancement notes, or other feature descriptions), these are all treated as queries in a location task based on IR techniques. In the case study in this chapter, most of features in the dataset are bug reports, and the remainders are patch and enhancement descriptions.

There are three main types of analysis used in feature location: dynamic, static and textual methods. Some studies have combined two or all of these techniques in order to obtain a better result than using each individual technique [1]. Dynamic methods gather information for analysis during the execution of programs, while static methods extract a structure such as a control or dependence graph from the source code to help developers analyze where the feature is located. Textual methods, i.e. IR techniques, analyze the text of the source code, including identifiers and comments. Rubin and Chechik [60] divided the types of feature location analysis into two categories, dynamic

and static, according to whether or not the program is executed as part of the location process.

The IR techniques used in feature location are based on utilizing the similarities between feature terms and source code artifacts to rank software entities according to their relevance to the feature query. Some IR models, such as LSI [12], [80] and LDA [2], [35], have been applied in the area of feature location. In this chapter, we focus solely on IR-based techniques, and compare the feature location performance of several IR methods. Many of the hybrid methods introduced in Section 2 combine textual methods with other information from historical data and results from static and dynamic analysis. These hybrid methods generally achieve better performance than each individual method alone. Choosing a suitable IR method for use in these hybrid techniques is an important issue in order to achieve good performance in feature location, since this is the core method in these techniques.

BM25 [52] is a popular ranking method for IR that uses two parameters to control the weights of term frequencies and document lengths. Its variant, BM25F [81], divides the documents into different fields and assigns different scaling parameters to the terms in each field. BM25 has previously been applied to duplication detection for software reports [82] and has been used as a weighting scheme in a structured source-code-based bug localization approach [4]. We conduct experiments on a feature location benchmark containing four open source projects [1] and show that the BM25 family of algorithms achieve the best results in comparison to three other IR methods in their best configurations.

The structure of this chapter is as follows. Related work that was not previously introduced in Chapter 2 is described in Section 5.2. We introduce feature location based on BM25 and its variant, BM25F, in Section 5.3. An experiment involving a case study is presented in Section 5.4. We present the results and a discussion in Section 5.5, and threats to the validity of this approach and conclusions are discussed in Sections 5.6 and 5.7.

5.2. Related Work

LSI [12], [80] and LDA [2], [35] have been applied in feature location. LSI [83] is an indexing and retrieval method that uses SVD to extract the latent semantic vectors from

the term-by-document matrix. The documents in the LSI model are represented as latent semantic vectors. Poshyvanyk and Marcus [84] used formal concept analysis (FCA) to organize the ranked list of source code entities from LSI-based feature location. LDA [23] is a topic model that assumes a latent topic level between documents and words. The LDA model uses topic distribution to represent the documents, where each topic is a multinomial distribution over terms.

Several papers [25], [37], [39], [85], [86] have shown that the performance of simple IR models such as VSM and UM are significantly better than sophisticated models such as LSI and LDA for feature location. Therefore, we choose VSM and UM as the main techniques for comparison. One reason for this result is that these studies used the entire feature description as the query, while LSI and LDA work on feature location using manually selected queries. This means that the input spaces of simple and sophisticated models are not the same.

In this chapter, we describe the use of BM25 and BM25F to locate features in software. The results of this experiment show that when all the terms of the feature descriptions are treated as queries, without automatic or manual reformulation, BM25 and BM25F achieve better performance than UM, VSM and LDA, while the UM and VSM models performed well when using only IR techniques in previous work [25], [37], [39], [85], [86].

BLUiR [4] is closely related to the work in this chapter. It improved bug localization using structured IR with the BM25 weighting scheme. The authors divided the source code terms into four fields, and the query terms into two fields, and calculated the similarity for each field combination, summing all scores to produce the final ranked list of source code entities.

Some hybrid techniques that combine IR methods such as LSI and LDA with other information such as the source code structure [13]–[15], dynamic analysis [10]–[12] and version history [7]–[9] have achieved better performance than IR methods alone. However, these hybrid methods typically need more input data rather than source code text. Sometimes, the additional input cannot provide enough information to improve the performance, for example, the version history of a newly developed system.

5.3. BM25-Based Feature Location

This section introduces approaches to feature location using BM25 and its variant, BM25F.

5.3.1. BM25

BM25 [52] is a bag-of-words ranking function implemented in the Okapi IR system. Each query term has a score that is based on the occurrence of this term in all the documents, regardless of the interrelationship between the query terms within a document. There are two parameters in BM25 that control the scale of both the term frequencies and document lengths. In this way, the influence of frequent terms and long documents on the final result can be calibrated based on the term frequencies and document lengths.

Given a query Q with terms $q_1, q_2, \dots, q_n$, the BM25 score of a document D is:

$$Score(D, Q) = \sum_{i=1}^n IDF(q_i) \frac{tf(q_i, D)(k_1 + 1)}{tf(q_i, D) + k_1 \left(1 - b + b \frac{|D|}{avgdl}\right)} \quad (5.1)$$

Here, $tf(q_i, D)$ is the term frequency of q_i in the document D ; $|D|$ is the length of document D ; and $avgdl$ is the average of the document lengths within the corpus as a whole. The term containing $|D|$, $avgdl$, and b represents the normalization of the document frequency. The free parameters k_1 and b control the scale of the term frequency and document length, respectively. For $k_1 = 0$, this is a binary model (i.e. does not consider term frequency). A b value of zero indicates no length normalization. $IDF(q_i)$ is the inverse document frequency (IDF) of the term q_i in the whole corpus:

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \quad (5.2)$$

Here, N is the total number of documents; $n(q_i)$ is the number of documents containing the term q_i ; and 0.5 is a smoothing constant for the situation $n(q_i) = 0$. The IDF value will be negative when $n(q_i)$ is greater than half of N . So the query term which has occurred in larger than half of all documents is ignored in the algorithm.

Since the feature queries in this case study are usually long documents, a normalizing scheme for the query term is used to deal with the term counts in a query, as follows [18]:

$$Score(D, Q) = \sum_{i=1}^n \frac{(k_q + 1)tf(q_i, Q)}{k_q + tf(q_i, Q)} IDF(q_i) \frac{tf(q_i, D)(k_1 + 1)}{tf(q_i, D) + k_1 \left(1 - b + b \frac{|D|}{avgdl}\right)} \quad (5.3)$$

where $tf(q_i, Q)$ is the term frequency of q_i in the query Q ; and k_q is the weighting parameter which calibrates the term frequency scaling of the query. The remainder of this equation is the same as in Eq. (5.1). When k_q is set to zero, the query normalizing term is equal to one. In long documents, terms with a higher $tf(q_i, Q)$ often play a more important role in the query, and the weight of this term therefore needs to be enhanced. A higher k_q value means that a frequent term in the query has a relatively high influence on the final similarity score. When the query is short, $tf(q_i, Q)$ is one or zero for most query terms, and it is unnecessary to use query term normalizing in this case. In our research, we tried several values for the three parameters k_l , b and k_q in order to obtain the best result.

After the scores between a query and all documents are derived using Eq. (5.3), each document is ranked according to the corresponding score for the query. Documents with high scores are ranked at the top of the final result, meaning that these documents are more relevant to the query.

5.3.2. BM25F

BM25F [81] is a variant of BM25 in which the terms in a document are classified into several fields with differing degrees of importance. The original BM25F divided the text in a webpage into the title text, the main body text and the anchor text.

A field-dependent normalized term frequency $tf'(t, f, D)$ of a term t in a field f of a document D is calculated as follows:

$$tf'(t, f, D) = \frac{tf(t, f, D)}{1 - b_f + b_f \frac{l(f, D)}{avgl(f)}} \quad (5.4)$$

where $tf(t, f, D)$ is the original term frequency of the term t occurring in the field f of the document D ; $l(f, D)$ is the total number of terms in the field f of the document D ; $avgl(f)$ is the average length of the field f for the corpus as a whole; and b_f is a field-dependent parameter for scaling field length such as b in BM25.

The final *pseudo-frequency* of term t in a document D is a linear weighted sum of the values of $tf'(t, f, D)$ from different fields:

$$tf'(t, D) = \sum_f w_f \cdot tf'(t, f, D) \quad (5.5)$$

where w_f is a field-dependent weight parameter. A larger value of w_f indicates a higher importance of the corresponding field.

A score for a query Q with terms $q_1, q_2, \dots, q_n$ for a document D is calculated as follows:

$$Score(D, Q) = \sum_{i=1}^n IDF(q_i) \frac{tf'(t, D)}{k_1 + tf'(t, D)} \quad (5.6)$$

where $IDF(q_i)$ is calculated using Eq. (5.2) in the same way as for BM25; and k_1 is the term frequency scaling parameter.

Since feature descriptions are often long, we use the same normalizing scheme for the query term as in Eq. (5.3) in BM25, as follows:

$$Score(D, Q) = \sum_{i=1}^n IDF(q_i) \frac{tf'(t, D)}{k_1 + tf'(t, D)} \frac{(k_q + 1)tf(q_i, Q)}{k_q + tf(q_i, Q)} \quad (5.7)$$

Finally, the documents are ranked according to the scores for a specific query, as for BM25.

For simplicity, we divide the source code into two fields for BM25F: method invocation text and other text. The method invocation field includes only terms from the invoked method names, and this is called the *invocation field*. The other terms in the class are classified into another field called the *main field*. This division of source code fields is a preliminary classification, and we leave experiments with more fields for future work. The parameters of both fields are named as shown in Table 5.1.

In addition to the four parameters in Table 5.1, there are two other parameters, k_1 and k_q . Since we only care about the relative value of the weighting parameter w_f rather than its absolute value, w_1 is set to one in all parameter settings, and different values of w_2 are used refer to different relative weights. Thus, in this two-field-based BM25F scheme, there are a total of five parameters that need to be determined before the algorithm is executed.

Table 5.1 Two-field BM25F parameters used in feature location

Field/Parameter	b_f	w_f
Main	b_1	w_1
Invocation	b_2	w_2

5.4. Case Study

We compare the performance of feature location methods using VSM, UM, LDA, BM25 and BM25F for four open source software systems. Simple methods such as VSM and UM have been reported to show a better performance than other sophisticated methods such as LDA for feature location. The results of this case study show that BM25 and BM25F outperform the other three methods, even for their best configuration.

5.4.1. Data Set

We used four open source softwares: ArogUML ⁹ , JabRef ¹⁰ , jEdit ¹¹ and muCommmander¹². The features, including bug reports, patch notes and enhancement descriptions of the four systems, and the gold set, which is a set of modified methods relevant to the given feature, are taken from benchmarks [1]available online¹³. Table 5.2 lists the size metrics for the four projects and the number of features for each system, where LOC stands for lines of code.

Table 5.2 Metrics of software systems

System	Version	LOC	Classes	Features
ArgoUML	0.22	131,020	1,474	91
JabRef	2.6b	74,138	590	39
jEdit	4.3	106,481	513	150
muCommander	0.8.5	76,649	1,069	92

Since not all the gold methods in the benchmark, which are those methods relevant to the specific query, exist in the modified version of software, we set the feature location granularity to class level, and all the gold methods are changed to classes. Then, each gold class can be found in the modified version of the software.

5.4.2. Preprocessing

JRipples [78] is an Eclipse plug-in that supports feature location. We modified JRipples to extract two fields: the terms of each class from the dataset for BM25F techniques, and the whole text of each class for the other four algorithms. Thomas et al. [25] recommend

⁹ <http://argouml.tigris.org/>

¹⁰ <http://jabref.sourceforge.net/>

¹¹ <http://www.jedit.org/>

¹² <http://www.mucommander.com/>

¹³ <http://www.cs.wm.edu/semeru/data/benchmarks/>

using all textual information from bug reports and software entities. In our experiments, bug titles, descriptions and text-based software entities, including identifiers, comments and so on, are considered as sources of text.

We removed a list of English stop words and Java key words. Terms with fewer than four characters or more than 14 characters were removed. We split multi-word terms based on camel case and underscores using the splitting tool in JRipples. The Porter stemmer [19] was used to obtain the term stems.

5.4.3. Setting

We used the R lda v1.3.2¹⁴ package to run the LDA models. We implemented VSM, UM, BM25 and BM25F using our own implementation in R.

a) *VSM*

We used the regular *tf-idf* weight in VSM as follows [18]:

$$weight(w_i, d) = \frac{tf(w_i, D)}{\max_j tf(w_j, D)} \log \frac{N}{df(w_i)} \quad (5.8)$$

Here, $tf(w_i, D)$ is the term frequency of term w_i in a document D , which is normalized by the maximum term frequency in the document D . N is the number of total documents, and $df(w_i)$ is the document frequency of term w_i , which is the number of documents containing the term w_i .

After the document is represented as a vector of the weight values for each term, the similarity between the query and the document is computed using an *overlap score measure*, which is the sum of the *tf-idf* weights of terms occurring in the query [18]:

$$Sim(Q, D) = \sum_{q_i \in Q} weight(q_i, d) \quad (5.9)$$

Since most of the queries contain long feature descriptions, we used a similar normalizing scheme as in Eq. (5.3):

¹⁴ <http://cran.r-project.org/web/packages/lda/index.html>

$$Sim(Q, D) = \sum_{q_i \in Q} weight(q_i, d) \frac{(k+1)tf(q_i, Q)}{k + tf(q_i, Q)} \quad (5.10)$$

In our experiment, the query scaling parameter k took values from zero to five, with an increment of 0.5.

b) UM

UM is a language model which employs a single multinomial distribution over the terms for each document. The probability of each term is calculated as the maximum likelihood of each term [18]. We used UM with Jelinek-Mercer smoothing [87] to avoid a zero probability of the query term, which does not occur in the documents. The probability of term w_i in document D with smoothing is calculated as follows:

$$P(w_i|D) = u \frac{tf(w_i, D)}{\sum_{j=1}^{|D|} tf(w_j, D)} + (1-u) \frac{tf(w_i, C)}{\sum_{j=1}^{|C|} tf(w_j, C)} \quad (5.11)$$

Here, $tf(w_i, D)$ is the term frequency of term w_i in the document D ; and $tf(w_i, C)$ is the term frequency of the term w_i in the whole corpus C . The denominators of the two terms in Eq. (5.11) are the total number of terms in the document D and the corpus C , respectively. The parameter u is used to weight the UM probability for the document and the whole corpus, and the value of u is set from 0 to 0.95 with an interval of 0.05 in our experiment. The similarity between the query and the document is the product of the likelihoods of each term in the query [18]:

$$Sim(Q, d_i) = P(Q|d_i) = \prod_{q_k \in Q} P(q_k|d_i) \quad (5.12)$$

where q_k is the k^{th} word in query Q ; and d_i represents the document. The query terms that do not occur in the whole document corpus are ignored.

c) LDA

The LDA [23] model has three parameters: the number of topics K , and two superparameters, α and β . Biggers et al. [88] have studied how the configurations of the parameters used in LDA influence the feature location performance for these four software systems. Since they studied configurations at method-level, which is different from our experiment, we tried several parameter values for LDA, as shown in Table 5.3.

The similarity between the query and the document is also calculated using Eq. (5.12), which followed the feature location approach based on LDA [2], [35]. The R-lda package is based on Gibbs sampling [89] to infer the model. We set the Gibbs sampling iteration to 2000 for all experiments to make the LDA model converge.

Table 5.3 LDA parameter settings

Parameter	Values
α	0.1, 0.25, 0.5, 1
β	0.1, 0.25, 0.5, 1.
K	50, 100, 150, 200

d) BM25 and BM25F

We also tried different combinations of parameter values for BM25 and BM25F. The BM25 and BM25F parameter settings are shown in Table 5.4 and Table 5.5, respectively. Since BM25F has more parameters than BM25, we set the parameters of BM25F to longer intervals than for BM25. The total number of parameter combinations for BM25 was 891, while the total number of parameter combinations for BM25F was 6300.

Table 5.4 BM25 parameter settings

Parameter	Values
k_1	1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5
b_1	0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1
k_q	1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

Table 5.5 BM25F parameter settings

Parameter	Values
k_1	1, 2, 3, 4, 5
b_1	0.5, 0.6, 0.7, 0.8, 0.9, 1
b_2	0.5, 0.6, 0.7, 0.8, 0.9, 1
w_1	1
w_2	0.1, 0.2, 0.5, 1, 2, 5, 10
k_q	1, 2, 3, 4, 5

5.5. Results and Analysis

The MRR results for the five methods, with the best four and worst four configurations, are shown in Table 5.6 and Table 5.7. The best configuration and the MRR metric for each method for each project are shown in bold type, and the rank is based on the MRR values. The parameter configurations are also ranked based on the MRR value. The MRR of the best configurations for BM25 and BM25F are similar for each project, and these are higher than the best configurations of UM, VSM and LDA for each project. A clearer comparison of the best configurations for each method and project is given in Figure 5.1.

As shown in Table 5.6, the parameter k in VSM has little influence on the performance. The parameter k in VSM takes values from 1.5 to 3.5 to obtain the best performance for the ArgoUML, JabRef and jEdit projects, while a value of $k = 0$ in muCommander performs best, meaning that the query term frequency is totally ignored. The performances of UM in different parameter configurations show that a large weight of document generating probability with a small weight of collection generating probability is reasonable in UM-based feature location. When $u = 0$, the scores of terms only count the collection generating probabilities with no document discrimination, so it is the worst configuration in UM.

The performance of the LDA model varies according to the different parameters. The hyperparameters settings of $\alpha = 1$, $\beta = 0.25$ are reasonable choices for all systems except for the muCommander project. 150 or 200 topic numbers exist mostly in the best four configurations. Since the inference procedure in LDA is probabilistically based iteration sampling, the result varies each time. The MRR results for LDA shown in Table 5.6 are calculated based on running the LDA model only once, but the 64 LDA results with different parameter values indicate that the performance of LDA is worse than the other methods, except for the JabRef project.

BM25 and BM25F have similar MRR values for feature location with the best and worst four configurations. This may be due to the fact that we consider only two simple fields for the source codes. The term frequency scaling parameters k_l in the two algorithms also have similar values for each project, as shown in Table 5.7. For example, $k_l = 4$ is suitable for both BM25 and BM25F for the jEdit project. Within the range of settings, a larger k_q value ($k_q = 4, 5$) is reasonable for both algorithms for four projects, indicating that certain terms in the query with large frequency need to be enhanced, since a larger

k_q value means the more frequent terms in the query have a larger influence on the final similarity score.

Table 5.6 The best and worst four configurations for VSM, UM, and LDA for feature location

VSM			UM			LDA			
Rank	Configuration	MRR	Rank	Configuration	MRR	Rank	Configuration	MRR	
ArgoUML									
1	$k = 1.5$	0.2382	1	$u = 0.9$	0.3271	1	$\alpha = 1, \beta = 0.5, K = 150$	0.2125	
2	$k = 1$	0.2357	2	$u = 0.95$	0.3244	2	$\alpha = 1, \beta = 0.25, K = 150$	0.1940	
3	$k = 2$	0.2336	3	$u = 0.85$	0.3204	3	$\alpha = 1, \beta = 0.1, K = 150$	0.1931	
4	$k = 4$	0.2305	4	$u = 0.8$	0.3149	4	$\alpha = 1, \beta = 0.5, K = 200$	0.1898	
8	$k = 0.5$	0.2275	17	$u = 0.15$	0.2356	61	$\alpha = 0.25, \beta = 1, K = 150$	0.0901	
9	$k = 3.5$	0.2264	18	$u = 0.1$	0.2281	62	$\alpha = 1, \beta = 0.1, K = 50$	0.0895	
10	$k = 5$	0.2253	19	$u = 0.05$	0.2138	63	$\alpha = 0.1, \beta = 1, K = 50$	0.0886	
11	$k = 0$	0.2226	20	$u = 0$	0.0014	64	$\alpha = 0.1, \beta = 1, K = 100$	0.0621	
JabRef									
1	$k = 3.5$	0.2438	1	$u = 0.8$	0.5763	1	$\alpha = 0.5, \beta = 0.25, K = 150$	0.3177	
2	$k = 5$	0.2412	2	$u = 0.85$	0.5702	2	$\alpha = 0.1, \beta = 0.1, K = 200$	0.2920	
3	$k = 4.5$	0.2397	3	$u = 0.7$	0.5606	3	$\alpha = 1, \beta = 0.1, K = 100$	0.2784	
4	$k = 3$	0.2378	4	$u = 0.75$	0.5580	4	$\alpha = 0.25, \beta = 0.1, K = 100$	0.2740	
8	$k = 1.5$	0.2331	17	$u = 0.15$	0.2357	61	$\alpha = 0.1, \beta = 1, K = 200$	0.0841	
9	$k = 1$	0.2186	18	$u = 0.1$	0.2281	62	$\alpha = 0.5, \beta = 1, K = 150$	0.0838	
10	$k = 0.5$	0.2085	19	$u = 0.05$	0.2138	63	$\alpha = 0.1, \beta = 1, K = 150$	0.0834	
11	$k = 0$	0.1906	20	$u = 0$	0.0014	64	$\alpha = 0.1, \beta = 0.5, K = 100$	0.0579	
jEdit									
1	$k = 2$	0.4778	1	$u = 0.8$	0.5488	1	$\alpha = 1, \beta = 0.25, K = 200$	0.3847	
2	$k = 2.5$	0.4732	2	$u = 0.75$	0.5467	2	$\alpha = 1, \beta = 0.1, K = 200$	0.3746	
3	$k = 3$	0.4691	3	$u = 0.85$	0.5431	3	$\alpha = 0.5, \beta = 0.1, K = 200$	0.3690	
4	$k = 3.5$	0.4688	4	$u = 0.9$	0.5427	4	$\alpha = 1, \beta = 0.1, K = 150$	0.3604	
8	$k = 1.5$	0.4628	17	$u = 0.15$	0.4300	61	$\alpha = 0.1, \beta = 1, K = 200$	0.2103	
9	$k = 5$	0.4619	18	$u = 0.1$	0.3997	62	$\alpha = 0.1, \beta = 1, K = 50$	0.2008	
10	$k = 0.5$	0.4462	19	$u = 0.05$	0.3836	63	$\alpha = 0.25, \beta = 1, K = 150$	0.1966	
11	$k = 0$	0.4262	20	$u = 0$	0.0039	64	$\alpha = 0.1, \beta = 1, K = 100$	0.1859	
muCommander									
1	$k = 0$	0.3765	1	$u = 0.85$	0.5174	1	$\alpha = 0.5, \beta = 0.1, K = 200$	0.3467	
2	$k = 0.5$	0.3645	2	$u = 0.9$	0.5170	2	$\alpha = 0.1, \beta = 0.1, K = 200$	0.3207	
3	$k = 1$	0.3617	3	$u = 0.95$	0.5112	3	$\alpha = 0.25, \beta = 0.1, K = 150$	0.3177	
4	$k = 2$	0.3580	4	$u = 0.8$	0.5082	4	$\alpha = 0.1, \beta = 0.1, K = 150$	0.3136	
8	$k = 4.5$	0.3557	17	$u = 0.15$	0.3822	61	$\alpha = 0.1, \beta = 1, K = 50$	0.1460	
9	$k = 3$	0.3537	18	$u = 0.1$	0.3510	62	$\alpha = 0.1, \beta = 1, K = 150$	0.1450	
10	$k = 3.5$	0.3515	19	$u = 0.05$	0.3188	63	$\alpha = 0.25, \beta = 1, K = 150$	0.1257	
11	$k = 2.5$	0.3515	20	$u = 0$	0.0019	64	$\alpha = 0.1, \beta = 1, K = 200$	0.1194	

Table 5.7 The best and worst four configurations for BM25 and BM25F for feature location

BM25			BM25F		
Rank	Configuration	MRR	Rank	Configuration ($w_l=1$)	MRR
ArgoUML					
1	$k_l = 1.5, b = 0.75, k_q = 4.5$	0.3577	1	$k_l = 3, b_l = 0.5, b_2 = 1, w_2 = 0.5, k_q = 4$	0.3561
2	$k_l = 1.5, b = 0.75, k_q = 5$	0.3556	2	$k_l = 3, b_l = 0.5, b_2 = 0.6, w_2 = 0.2, k_q = 4$	0.3560
3	$k_l = 1, b = 0.9, k_q = 2.5$	0.3549	3	$k_l = 3, b_l = 0.5, b_2 = 0.5, w_2 = 0.2, k_q = 4$	0.3560
4	$k_l = 3, b = 0.55, k_q = 5$	0.3522	4	$k_l = 3, b_l = 0.5, b_2 = 0.7, w_2 = 0.2, k_q = 4$	0.3560
888	$k_l = 5, b = 1, k_q = 4.5$	0.2416	6297	$k_l = 5, b_l = 1, b_2 = 0.9, w_2 = 5, k_q = 4$	0.2221
889	$k_l = 5, b = 1, k_q = 5$	0.2416	6298	$k_l = 5, b_l = 1, b_2 = 1, w_2 = 5, k_q = 4$	0.2221
890	$k_l = 4.5, b = 1, k_q = 1$	0.2385	6299	$k_l = 5, b_l = 1, b_2 = 0.9, w_2 = 5, k_q = 1$	0.2210
891	$k_l = 5, b = 1, k_q = 1$	0.2353	6300	$k_l = 5, b_l = 1, b_2 = 0.8, w_2 = 5, k_q = 1$	0.2271
JabRef					
1	$k_l = 1, b = 0.65, k_q = 4.5$	0.6124	1	$k_l = 3, b_l = 0.5, b_2 = 0.5, w_2 = 5, k_q = 4$	0.6369
2	$k_l = 1, b = 0.75, k_q = 4$	0.6013	2	$k_l = 3, b_l = 0.5, b_2 = 0.1, w_2 = 10, k_q = 4$	0.6368
3	$k_l = 1, b = 0.8, k_q = 5$	0.6013	3	$k_l = 3, b_l = 0.5, b_2 = 0.9, w_2 = 10, k_q = 4$	0.6368
4	$k_l = 1, b = 0.6, k_q = 4.5$	0.6005	4	$k_l = 3, b_l = 0.5, b_2 = 0.7, w_2 = 10, k_q = 4$	0.6347
888	$k_l = 4, b = 1, k_q = 1$	0.4286	6297	$k_l = 5, b_l = 1, b_2 = 0.5, w_2 = 1, k_q = 1$	0.4167
889	$k_l = 5, b = 0.95, k_q = 1$	0.4267	6298	$k_l = 5, b_l = 1, b_2 = 0.8, w_2 = 2, k_q = 1$	0.4166
890	$k_l = 4.5, b = 1, k_q = 1$	0.4192	6299	$k_l = 5, b_l = 1, b_2 = 0.9, w_2 = 2, k_q = 1$	0.4166
891	$k_l = 5, b = 1, k_q = 1$	0.4167	6300	$k_l = 5, b_l = 1, b_2 = 1, w_2 = 2, k_q = 1$	0.4166
jEdit					
1	$k_l = 4, b = 0.55, k_q = 4.5$	0.6376	1	$k_l = 4, b_l = 0.5, b_2 = 0.5, w_2 = 0.5, k_q = 5$	0.6373
2	$k_l = 4, b = 0.55, k_q = 5$	0.6367	2	$k_l = 4, b_l = 0.5, b_2 = 0.8, w_2 = 1, k_q = 5$	0.6373
3	$k_l = 4.5, b = 0.5, k_q = 5$	0.6366	3	$k_l = 4, b_l = 0.5, b_2 = 0.8, w_2 = 0.5, k_q = 5$	0.6373
4	$k_l = 4, b = 0.55, k_q = 4$	0.6363	4	$k_l = 4, b_l = 0.5, b_2 = 1, w_2 = 0.5, k_q = 5$	0.6373
888	$k_l = 3.5, b = 1, k_q = 1$	0.5506	6297	$k_l = 4, b_l = 1, b_2 = 0.6, w_2 = 10, k_q = 1$	0.5203
889	$k_l = 5, b = 1, k_q = 1$	0.5488	6298	$k_l = 5, b_l = 1, b_2 = 0.7, w_2 = 10, k_q = 1$	0.5179
890	$k_l = 4.5, b = 1, k_q = 1.5$	0.5484	6299	$k_l = 5, b_l = 1, b_2 = 0.8, w_2 = 10, k_q = 1$	0.5174
891	$k_l = 4.5, b = 1, k_q = 1$	0.5431	6300	$k_l = 5, b_l = 1, b_2 = 0.9, w_2 = 10, k_q = 1$	0.5170
muCommander					
1	$k_l = 3, b = 0.55, k_q = 5$	0.5743	1	$k_l = 4, b_l = 0.5, b_2 = 0.5, w_2 = 2, k_q = 3$	0.5760
2	$k_l = 4, b = 0.5, k_q = 2.5$	0.5737	2	$k_l = 4, b_l = 0.5, b_2 = 0.6, w_2 = 2, k_q = 3$	0.5741
3	$k_l = 4.5, b = 0.5, k_q = 2.5$	0.5727	3	$k_l = 4, b_l = 0.5, b_2 = 0.8, w_2 = 2, k_q = 3$	0.5732
4	$k_l = 2.5, b = 0.6, k_q = 4.5$	0.5600	4	$k_l = 4, b_l = 0.5, b_2 = 0.7, w_2 = 2, k_q = 3$	0.5732
888	$k_l = 5, b = 1, k_q = 3.5$	0.4425	6297	$k_l = 5, b_l = 1, b_2 = 0.5, w_2 = 2, k_q = 1$	0.4345
889	$k_l = 5, b = 1, k_q = 5$	0.4413	6298	$k_l = 5, b_l = 1, b_2 = 0.7, w_2 = 2, k_q = 1$	0.4343
890	$k_l = 5, b = 1, k_q = 1.5$	0.4390	6299	$k_l = 5, b_l = 1, b_2 = 0.6, w_2 = 2, k_q = 1$	0.4342
891	$k_l = 5, b = 1, k_q = 1$	0.4370	6300	$k_l = 5, b_l = 1, b_2 = 1, w_2 = 5, k_q = 4$	0.4341

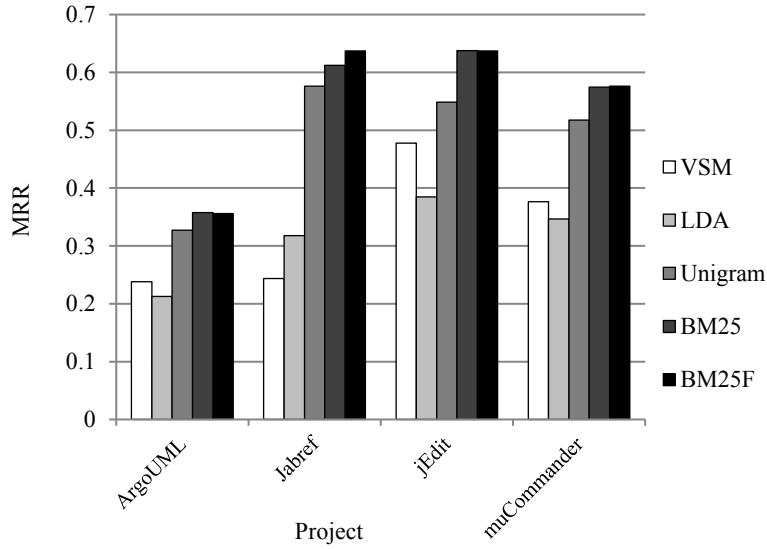


Figure 5.1 MRR for IR techniques with best configurations

In BM25, $b = 1$; this means that the full scaling of the document length exists in the worst four configurations of all four projects, indicating that full document length normalization is not suitable for this case study. In BM25F, $b_l = 0.5$, the lowest value of the corresponding setting range, is shown in all four of the best configurations for all projects, while $b_l = 1$, the highest value of the setting range, is shown in all four of the worst configurations for all projects. This indicates that for the main field, a low level of scaling for the field length is sufficient. Since we did not test a value of b_l lower than 0.5, future work is needed to focus on a larger range of parameter configurations. Suitable values of the field weight parameter w_2 are different for different projects. There is no obvious trend showing that the method invocation field should be weighted higher or lower than the main field.

5.6. Threats to Validity

There are several threats to the validity of our results.

Firstly, we used several parameter values for each IR method in our case study; however, in practice, we cannot set these parameters without knowing the relevant entities of the queries. Some reasonable parameter settings for these five methods for our selected four systems were discussed in the previous section, but we cannot ensure that these are suitable for other software systems.

Although BM25 and BM25F achieve better performance than the other three methods in each best configuration for the five methods, parameter tuning for the BM25 family of algorithms is still a challenge, even if we know the previous features and relevant entities of a system. For BM25F, one choice is to use the original BM25F parameter optimizing procedure [81]; this is a local optimization process and needs to know the relevant entities of features in advance. Empirical parameter recommendations for the use of the BM25 algorithm in feature location should be studied in future work.

The evaluation metric used here, MRR, is based on the effective measure, which is the rank of the first relevant entity in the returned list. Since there are typically multiple entities that are relevant to a given feature, the ranks of the other entities besides the top ranked one are not considered in this evaluation metric. Nevertheless, this metric is widely used in feature location and allows different techniques to be compared with each other.

Since the amount of computation required is relatively large, we did not use a wider range of parameter values in our case study. Some parameters were set to the smallest or largest values to obtain the best or worst performance. The use of more parameter values may generate more reasonable results.

5.7. Summary

Manual searching of software entities related to specific functionalities costs a great deal of time. Several IR-based feature location approaches have been proposed to help developers to find the initial location of relevant features. This chapter proposes feature location approaches based on BM25 and BM25F, and shows that these are better than three other standard IR methods, namely VSM, UM, and LDA, for four open source projects. There is still a great deal of work to do in this area in the future. We used only MRR as the evaluation metric, and ranked the results based on the MRR scores; other metrics could be used in future work, such as Top-N ranking, which indicates the number of features whose relevant documents are ranked in the top N items of the return lists, and MAP, which is the mean of the average precision scores for each query. We classified textual terms into method invocations and other terms, and used these as two fields in our BM25F-based feature location approach. The definition of more fields in BM25F may improve the performance. For example, text-based terms in class/method names, comments, identifiers, or other source code text could be considered as

individual fields in BM25F. Certain existing hybrid feature location methods combining IR models with other information may use the BM25 family of algorithms as an IR method, in order to improve performance.

Chapter 6. Conclusions and Future Work

In this chapter, we first conclude the whole thesis and then introduce some directions for the future work.

6.1. Conclusions

The software development and maintenance process generate features in the requirements document and bugs in bug reports. To help developers find the initial location of features and bugs in source code more efficiently, prior researchers have proposed various location approaches. Of these, IR-based approaches need only the textual information of features/bugs and the source code entities and can provide a ranked list of source code entities, mainly depending on the textual similarity between the feature description (bug report) and source code entities. However, these techniques have several limitations and associated problems. In this thesis, we introduce three ways to increase the performance of bug localization and feature location, from three different perspectives.

1) We introduce a new strategy that can determine a suitable point in time to stop using automatic bug localization methods and to switch to conventional localization approaches instead. This is because some bugs cannot be localized effectively using automatic methods, although there are many other bugs that generate highly ranked relevant entities in the list of results using an automatic method, which we refer to as easy bugs. Conventional localization approaches typically involve a higher cost to developers in terms of time to localize these easy bugs compared to automatic methods but require less time to localize hard bugs. For this reason, switching from automatic methods to conventional approaches is a feasible way to increase the performance, when the automatic methods are found to be ineffective for the current bug. The proposed strategy optimizes the time at which the developer should discontinue the use of an automatic method and switch to conventional approaches.

2) Apart from the textual similarity, many other attributes are found to be beneficial to bug localization. These attributes are derived from various sources, including the version

history, dependence graph, text structure, stack traces and so on. However, there is little research focusing on the combining method in hybrid bug localization. We select beneficial attributes from different six types and combine these attributes using eight different LtR techniques, which are ranking algorithms drawn from the training set in IR. The results show that combinations of these selected attributes are more effective in localizing bugs than two state-of-the-art approaches which use only partial attributes. We also find suitable LtR techniques as the combining method in our case study of bug localization.

3) BM25 and its variant, BM25F, are popular IR techniques. We compare feature location based on BM25 and BM25F with conventional IR models, including VSM, SUM, and LDA. Although these models need to be tuned to determine the parameter values, well-tuned BM25 and BM25F algorithms perform better than well-tuned VSM, SUM and LDA approaches. We divide the source code text into two fields when applying BM25F in feature location, and find that BM25F performs slightly better than BM25 in our setting.

In conclusion, this paper presents three methods based on different aspects of an IR-based bug localization and feature location framework. Experiments show that each of these aspects is efficient and effective in real bug localization and feature location datasets. Additionally, these three methods can be combined within an organic whole to improve the performance of bug localization and feature location.

6.2. Future work

Some directions of future work for feature location and bug localization are presented in the following:

1) In the proposed strategy switching from automatic bug localization to conventional methods, we used many mathematical assumptions to simplify the model. Especially, the expected time cost of a conventional method is set to a constant value. However, this time cost is hard to expect in a real software project. In order to make the strategy more effectively, the research on the expected time cost of conventional approaches is very important. Recent work utilizes the time period between the bug open date and fixed date to predict the time cost, but this approach is not precise. For example, a bug may cost one month to fix, but the developers may only spend several hours in this fixing

issue. So the precise prediction of the time cost using conventional localization methods is necessary for applying the proposed strategy.

2) When comparing different LtR algorithms as the combining method in hybrid bug localization, the selected attributes or features are very important to the precision of the localization method, as well as the conclusion that ordinate ascent is the most suitable LtR algorithms for combining attributes in bug localization. In the experiment, we use as many features as possible, which have been used in bug localization. However, it is unavoidable that our selected features cannot involve all suitable features. Additionally, we have not applied the feature selection for those used features, but give the feature selection work to the LtR algorithms. All the above problems need to be studied in the future, to ensure the validity of the conclusion.

Bibliography

- [1] B. Dit, M. Reville, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *J. Softw. Evol. Process*, vol. 25, no. 1, pp. 53–95, 2013.
- [2] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation," in *15th Working Conference on Reverse Engineering*, 2008, pp. 155–164.
- [3] S. Davies, M. Roper, and M. Wood, "Using Bug Report Similarity to Enhance Bug Localisation," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 125–134.
- [4] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013, pp. 345–355.
- [5] S. Wang and D. Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, 2014, pp. 53–63.
- [6] Z. Shi, J. Keung, K. E. Bennin, N. Limsettho, and Q. Song, "A Strategy to Determine When to Stop Using Automatic Bug Localization," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016, vol. 1, pp. 185–190.
- [7] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 14–24.
- [8] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where Should We Fix This Bug? A Two-Phase Recommendation Model," *Softw. Eng. IEEE Trans.*, vol. 39, no. 11, pp. 1597–1610, Nov. 2013.
- [9] B. Sisman and A. C. Kak, "Incorporating version histories in Information Retrieval based bug localization," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, 2012, pp. 50–59.
- [10] B. Dit, M. Reville, and D. Poshyvanyk, "Integrating information retrieval, execution and link analysis algorithms to improve feature location in software," *Empir. Softw. Eng.*, vol. 18, no. 2, pp. 277–309, 2013.
- [11] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 234–243.
- [12] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *Softw. Eng. IEEE Trans.*, vol. 33, no. 6, pp. 420–432, 2007.
- [13] N. Ali, A. Sabane, Y. Gueheneuc, and G. Antoniol, "Improving Bug Location Using Binary Class Relationships," in *Source Code Analysis and*

- Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, 2012, pp. 174–183.
- [14] G. Scanniello and A. Marcus, “Clustering Support for Static Concept Location in Source Code,” in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, 2011, pp. 1–10.
 - [15] P. Singh and S. Batra, “A Novel Technique for Call Graph Reduction for Bug Localization,” *Int. J. Comput. Appl.*, vol. 47, no. 15, pp. 1–5, 2012.
 - [16] T.-Y. Liu, *Learning to Rank for Information Retrieval*. Springer Berlin Heidelberg, 2011.
 - [17] Z. Shi, J. Keung, and Q. Song, “An Empirical Study of BM25 and BM25F Based Feature Location Techniques,” in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, 2014, pp. 106–114.
 - [18] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.
 - [19] M. F. Porter, “An algorithm for suffix stripping,” *Progr. Electron. Libr. Inf. Syst.*, vol. 14, no. 3, pp. 130–137, 1980.
 - [20] G. Salton, A. Wong, and C. S. Yang, “A Vector Space Model for Automatic Indexing,” *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.
 - [21] C. Zhai and J. Lafferty, “A Study of Smoothing Methods for Language Models Applied to Information Retrieval,” *ACM Trans. Inf. Syst.*, vol. 22, no. 2, pp. 179–214, 2004.
 - [22] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification,” in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 137–148.
 - [23] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet Allocation,” *J. Mach. Learn. Res.*, vol. 3, no. 4–5, pp. 993–1022, 2003.
 - [24] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, “Can Better Identifier Splitting Techniques Help Feature Location?,” in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, 2011, pp. 11–20.
 - [25] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, “The Impact of Classifier Configuration and Classifier Combination on Bug Localization,” *Softw. Eng. IEEE Trans.*, vol. 39, no. 10, pp. 1427–1443, 2013.
 - [26] V. Dallmeier and T. Zimmermann, “Extraction of Bug Localization Benchmarks from History,” in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 433–436.
 - [27] R. Beltrán-Alfonso, A. Torres-Tautiva, P. A. Gaona-García, and C. E. Montenegro-Marin, “Exploring the Relevance of Search Engines: An Overview of Google as a Case Study,” *Int. J. Interact. Multimed. Artif. Intell.*, vol. 4, no. 4, pp. 72–79, 2017.
 - [28] E. M. Voorhees, “The TREC-8 Question Answering Track Report,” in *Natural Language Engineering*, vol. 7, no. 04, E. Voorhees and De. Harman, Eds. NIST, 1999, pp. 77–82.
 - [29] M. Beard, N. Kraft, L. Etzkorn, and S. Lukins, “Measuring the Accuracy of Information Retrieval Based Bug Localization Techniques,” in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 124–128.

- [30] E. Hill, A. Bacchelli, D. Binkley, B. Dit, D. Lawrie, and R. Oliveto, "Which Feature Location Technique is Better?," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 408–411.
- [31] C. Tantithamthavorn, A. Ihara, H. Hata, and K. Matsumoto, "Impact Analysis of Granularity Levels on Feature Location Technique," in *Requirements Engineering*, vol. 432, D. Zowghi and Z. Jin, Eds. Springer Berlin Heidelberg, 2014, pp. 135–149.
- [32] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 2014, pp. 181–190.
- [33] R. Abreu, P. Zoetewij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.
- [34] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 273–282.
- [35] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 972–990, 2010.
- [36] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 118–121.
- [37] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus, "On the Relationship between the Vocabulary of Bug Reports and Source Code," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 452–455.
- [38] B. D. Nichols, "Augmented Bug Localization Using Past Bug Information," in *Proceedings of the 48th Annual Southeast Regional Conference*, 2010, p. 61:1--61:6.
- [39] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.
- [40] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011, pp. 263–272.
- [41] S. Lal and A. Sureka, "A Static Technique for Fault Localization Using Character N-gram Based Information Retrieval Model," in *Proceedings of the 5th India Software Engineering Conference*, 2012, pp. 109–118.
- [42] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, 2013, pp. 247–256.
- [43] C. Tantithamthavorn, A. Ihara, and K.-I. Matsumoto, "Using Co-change Histories to Improve Bug Localization Performance," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, 2013, pp. 543–548.
- [44] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the Use of Stack

- Traces to Improve Text Retrieval-Based Bug Localization,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 151–160.
- [45] X. Ye, R. Bunescu, and C. Liu, “Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 689–699.
 - [46] T.-D. B. Le, R. J. Oentaryo, and D. Lo, “Information Retrieval and Spectrum Based Bug Localization: Better Together,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.
 - [47] T. Dilshener, M. Wermelinger, and Y. Yu, “Locating Bugs Without Looking Back,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 286–290.
 - [48] S. Wang and D. Lo, “AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization,” *J. Softw. Evol. Process*, vol. 28, no. 10, pp. 921–942, 2016.
 - [49] X. Ye, R. Bunescu, and C. Liu, “Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 4, pp. 379–402, Apr. 2016.
 - [50] K. C. Youm, J. Ahn, and E. Lee, “Improved bug localization based on code change histories and bug reports,” *Inf. Softw. Technol.*, vol. 82, pp. 177–192, 2017.
 - [51] T. Joachims, “Training Linear SVMs in Linear Time,” in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 217–226.
 - [52] S. J. M. M. H.-B. S. E. Robertson S. Walker and M. Gatford, “Okapi at TREC-3,” in *Overview of the Third Text REtrieval Conference (TREC)*, 1995.
 - [53] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Comput. Networks {ISDN} Syst.*, vol. 30, no. 1–7, pp. 107–117, 1998.
 - [54] J. M. Kleinberg, “Authoritative Sources in a Hyperlinked Environment,” *J. ACM*, vol. 46, no. 5, pp. 604–632, 1999.
 - [55] F. Thung, T.-D. B. Le, P. S. Kochhar, and D. Lo, “BugLocalizer: Integrated Tool Support for Bug Localization,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 767–770.
 - [56] C. Lewis and R. Ou, “Bug prediction at google,” 2011.
 - [57] K. Chen and V. Rajlich, “Case study of feature location using dependence graph,” in *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, 2000, pp. 241–247.
 - [58] G. Antoniol and Y.-G. Gueheneuc, “Feature Identification: An Epidemiological Metaphor,” *Softw. Eng. IEEE Trans.*, vol. 32, no. 9, pp. 627–641, 2006.
 - [59] T. Dao, L. Zhang, and N. Meng, “How Does Execution Information Help with Information-retrieval Based Bug Localization?,” in *Proceedings of the 25th International Conference on Program Comprehension*, 2017, pp. 241–250.
 - [60] J. Rubin and M. Chechik, “A Survey of Feature Location Techniques,” in

- Domain Engineering*, Springer Berlin Heidelberg, 2013, pp. 29–58.
- [61] J. Wang, X. Peng, Z. Xing, and W. Zhao, “How developers perform feature location tasks: a human-centric and process-oriented exploratory study,” *J. Softw. Evol. Process*, vol. 25, no. 11, pp. 1193–1224, 2013.
 - [62] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, “What Makes a Good Bug Report?,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 308–318.
 - [63] K. Herzig, S. Just, and A. Zeller, “It’s Not a Bug, It’s a Feature: How Misclassification Impacts Bug Prediction,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 392–401.
 - [64] Q. Wang, C. Parnin, and A. Orso, “Evaluating the Usefulness of IR-based Fault Localization Techniques,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 1–11.
 - [65] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of Test Information to Assist Fault Localization,” in *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 467–477.
 - [66] H. Zhang, L. Gong, and S. Versteeg, “Predicting Bug-fixing Time: An Empirical Study of Commercial Software Projects,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 1042–1051.
 - [67] D. Binkley and D. Lawrie, “Learning to Rank Improves IR in SE,” in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 2014, pp. 441–445.
 - [68] D. Metzler and W. Bruce Croft, “Linear feature-based models for information retrieval,” *Inf. Retr. Boston.*, vol. 10, no. 3, pp. 257–274, 2007.
 - [69] F. Zhao, Y. Tang, Y. Yang, H. Lu, Y. Zhou, and B. Xu, “Is Learning-to-Rank Cost-Effective in Recommending Relevant Files for Bug Localization?,” in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 298–303.
 - [70] L. Breiman, “Random Forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
 - [71] J. H. Friedman, “Greedy Function Approximation: A Gradient Boosting Machine,” *Ann. Stat.*, vol. 29, no. 5, pp. 1189–1232, 2001.
 - [72] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, “Learning to Rank Using Gradient Descent,” in *Proceedings of the 22Nd International Conference on Machine Learning*, 2005, pp. 89–96.
 - [73] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, “An Efficient Boosting Algorithm for Combining Preferences,” *J. Mach. Learn. Res.*, vol. 4, pp. 933–969, 2003.
 - [74] Q. Wu, C. C. Burges, K. Svore, and J. Gao, “Adapting boosting for information retrieval measures,” *Inf. Retr. Boston.*, vol. 13, no. 3, pp. 254–270, 2010.
 - [75] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li, “Learning to Rank: From Pairwise Approach to Listwise Approach,” in *Proceedings of the 24th International Conference on Machine Learning*, 2007, pp. 129–136.
 - [76] J. Xu and H. Li, “AdaRank: A Boosting Algorithm for Information Retrieval,” in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007, pp. 391–398.

- [77] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting Structural Information from Bug Reports," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, 2008, pp. 27–30.
- [78] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A Tool for Program Comprehension During Incremental Change," in *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 149–152.
- [79] P. S. Kochhar, Y. Tian, and D. Lo, "Potential Biases in Bug Localization: Do They Matter?," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014, pp. 803–814.
- [80] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004, pp. 214–223.
- [81] H. Zaragoza, H. Zaragoza, N. Craswell, N. Craswell, M. Taylor, M. Taylor, S. Saria, S. Saria, S. Robertson, and S. Robertson, "Microsoft Cambridge at TREC-13: Web and HARD tracks," *Proc. TREC 2004*, 2004.
- [82] C.-Z. Yang, H.-H. Du, S.-S. Wu, and I.-X. Chen, "Duplication Detection for Software Bug Reports Based on BM25 Term Weighting," in *Technologies and Applications of Artificial Intelligence (TAAI), 2012 Conference on*, 2012, pp. 33–38.
- [83] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Am. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.
- [84] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 2007, pp. 37–48.
- [85] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The Effect of Lexicon Bad Smells on Concept Location in Source Code," in *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, 2011, pp. 125–134.
- [86] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 92–96.
- [87] F. JELINEK, "Interpolated estimation of Markov source parameters from sparse data," *Pattern Recognit. Pract.*, 1980.
- [88] L. Biggers, C. Bocovich, R. Capshaw, B. Eddy, L. Etzkorn, and N. Kraft, "Configuring latent Dirichlet allocation based feature location," *Empir. Softw. Eng.*, pp. 1–36, 2012.
- [89] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. Natl. Acad. Sci.*, vol. 101, no. Supplement 1, pp. 5228–5235, 2004.

List of publication

Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Zhang Xinjun, “Comparing Learning to Rank Techniques in Hybrid Bug Localization,” *Applied Soft Computing*. vol. 62, pp. 636–648, Jan. 2018.

Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, Nachai Limsettho and Qinbao Song, “A Strategy to Determine When to Stop Using Automatic Bug Localization,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016, vol. 1, pp. 185–190.

Zhendong Shi, Jacky Keung and Qinbao Song, “An Empirical Study of BM25 and BM25F Based Feature Location Techniques,” in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, 2014, pp. 106–114.