

DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

EFFECTIVE SOFTWARE BUG LOCALIZATION
USING INFORMATION RETRIEVAL AND
MACHINE LEARNING ALGORITHMS

by
Mustafa ERŞAHİN

December, 2020
İZMİR

EFFECTIVE SOFTWARE BUG LOCALIZATION USING INFORMATION RETRIEVAL AND MACHINE LEARNING ALGORITHMS

**A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Doctor of
Philosophy in Computer Engineering**

**by
Mustafa ERŞAHİN**

**December, 2020
İZMİR**

Ph.D. THESIS EXAMINATION RESULT FORM

We have read the thesis entitled “EFFECTIVE SOFTWARE BUG LOCALIZATION USING INFORMATION RETRIEVAL AND MACHINE LEARNING ALGORITHMS” completed by MUSTAFA ERŞAHİN under supervision of ASSOC. PROF. DR. SEMİH UTKU and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Assoc Prof. Dr. Semih UTKU

Supervisor

Prof. Dr. Alp KUT

Thesis Committee Member

Assoc. Prof. Dr. Derya Eren AKYOL

Thesis Committee Member

Assoc. Prof. Dr. Deniz KILINÇ

Examining Committee Member

Assoc. Prof. Dr. Gıyasettin ÖZCAN

Examining Committee Member

Prof. Dr. Özgür ÖZÇELİK

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisors, Assoc. Prof. Dr. Semih UTKU and Assoc. Prof. Dr. Deniz KILINC for their support, patient guidance, supervision and useful suggestions throughout this study. Their guidance helped me in all the time of research and writing of this thesis.

Mustafa ERŞAHİN

EFFECTIVE SOFTWARE BUG LOCALIZATION USING INFORMATION RETRIEVAL AND MACHINE LEARNING ALGORITHMS

ABSTRACT

Software quality assurance is crucial for the success of software. In large scale software projects, bug localization is a difficult and costly process. Many issues or bugs may be reported at both the development and maintenance phases of the software development lifecycle. Bug fixing has an essential role in software quality assurance, and bug localization is the first step of this process. Bug localization (BL) is time-consuming since the developers should understand the flow, coding structure, and logic of the program. Hence, it is crucial for developers to discover the location of the bug. In general, source codes and bug reports are used for identifying bug location with the help of many different techniques. Information retrieval-based bug localization (IRBL) also uses the information of bug reports and source code to locate the section of code in which the bug occurs. It is not possible to apply state-of-the-art approaches having a satisfactory performance to new projects according to the diversity of software development languages, design patterns and development standards.

This thesis proposes a novel algorithm, Adaptive Attribute Weighting (AAW), to adopt a new software project for BL processes. This thesis also includes the development of a new BL tool, BugSTAiR, in order to work on with all software projects. Experimental studies demonstrate the capability of the AAW algorithm and also the BugSTAiR tool on both real-life and experimental datasets, including commercial software projects which are developed with different languages and improvement in performance compared to the existing algorithms.

Keywords: Software engineering, bug localization, genetic algorithm, information retrieval, software process improvement

BİLGİ GERİ GETİRİMİ VE MAKİNE ÖĞRENMESİ ALGORİTMALARINI KULLANARAK YAZILIMDA HATA KONUMLANDIRILMASI

ÖZ

Yazılım kalite güvence yönetimi, yazılımın başarısı için çok önemlidir. Büyük ölçekli yazılım projelerinde, hata konumlandırma zor ve maliyetli bir süreçtir. Yazılım geliştirme yaşam döngüsünün hem geliştirme hem de bakım aşamasında birçok sorun veya hata rapor edilebilir. Hata düzeltmenin yazılım kalite güvencesinde önemli bir rolü vardır ve hata konumlandırma bu sürecin ilk adımıdır. Geliştiriciler programın akışını, kodlama yapısını ve mantığını anlaması gerektiği için zaman alıcı bir süreçtir. Bu nedenle, geliştiricilerin hatanın yerini keşfetmeleri önemlidir. Genel olarak, yazılım kaynak kodları ve hata kayıtları, farklı teknikler yardımıyla hata konumunun tanımlanması için kullanılır. Bilgi geri getirme tabanlı hata konumlandırma, hatanın olduğu kod bölümünü bulmak için hata raporları ve kaynak kodu bilgilerini kullanır. Bilinen en iyi yöntemler, yazılım geliştirme dillerinin çeşitliliğine, tasarım örüntülerinin ve geliştirme standartlarının farklılıklarından dolayı yeni projelerde tatmin edici bir performans gösterecek şekilde uygulanmaları mümkün değildir.

Bu tez, hata konumlandırma süreçlerini yeni bir yazılım projesine uyarlamak için yeni bir algoritma olan Adaptasyonlu Ağırlık Belirleme'yi önermektedir. Bu tez aynı zamanda tüm yazılım projeleri üzerinde çalışmak için yeni bir hata konumlandırma aracı olan BugSTAiR'nin geliştirilmesini de içermektedir. Deneysel çalışmalar, BugSTAiR aracının Adaptasyonlu Ağırlık Belirleme algoritmasının etkisiyle, farklı dillerle geliştirilen ticari yazılım projelerinde, hem gerçek yaşam hem de deneysel veri kümeleri dahil olmak üzere, mevcut algoritmalara göre performansının üst seviyede olduğunu göstermektedir.

Anahtar kelimeler: Yazılım mühendisliği, hata konumlandırma, genetik algoritma, bilgi geri getirme, yazılım süreç iyileştirme

CONTENTS

	Page
Ph.D. THESIS EXAMINATION RESULT FORM.....	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZ.....	v
LIST OF FIGURES.....	ix
LIST OF TABLES	x
CHAPTER 1 – INTRODUCTION.....	1
1.1 General.....	1
1.2 Purpose	2
1.3 Novel Contributions of this Thesis	3
1.4 Organization of the Thesis.....	3
CHAPTER 2 – RELATED WORK.....	5
2.1 Literature Review	5
2.2 Field Review.....	8
CHAPTER 3 – BACKGROUND INFORMATION	9
3.1 Bug Localization.....	9
3.2 Deep Learning-Based Approaches to Bug Localization.....	10
3.3 Information-Based Approaches to Bug Localization	11
3.3.1 Common Bug Localization Process	12
3.3.2 BugLocator.....	13
3.3.4 BLUIR	15
3.3.5 BLIA	16

CHAPTER 4 – ADAPTIVE ATTRIBUTE WEIGHTING ALGORITHM.....	18
4.1 Adaptation Process	18
4.1.1 IRBL Processes in BugSTAiR	19
4.1.2 Source Code – Bug Report Pre-processing	22
4.1.3 Indexing	23
4.1.4 Query Construction	24
4.1.5 In-memory Source Index (IMSI)	26
4.1.6 AAW	26
CHAPTER 5 – A NEW BUG LOCALIZATION TOOL: BugSTAiR	34
5.1 BugSTAiR	34
5.1.1 BugSTAiR Core Services	34
5.1.2 BugSTAiR UI Dashboard	36
CHAPTER 6 – EXPERIMENTAL STUDY	42
6.1 Subject Systems	42
6.1.1 Dataset Statistics	43
6.1.2 Evaluation Metrics	45
6.1.3 Experimental Results	46
6.1.4 Threats to Validity	50
CHAPTER 7 – CONCLUSION AND FUTURE WORK	52
7.1 Conclusion	52
7.2 Future Work	53
REFERENCES	55

APPENDICES.....	58
------------------------	-----------

APPENDIX: LIST OF ACRONYMS	58
----------------------------------	----



LIST OF FIGURES

	Page
Figure 3.1 The overall architecture of DeepLoc	11
Figure 3.2 Bug summary of a real-world application.....	12
Figure 3.3 General view of IRBL processes	13
Figure 3.4 The overall architecture of BugLocator	14
Figure 3.5 The overall architecture of BLUIR	15
Figure 3.6 BLIA's workflow	16
Figure 4.1 General view of BugSTAiR architecture	19
Figure 5.1 Login Page of BugSTAiR	39
Figure 5.2 Home page of BugSTAiR	39
Figure 5.3 Previous search list of a user.....	40
Figure 5.4 File search and search results according to file score	40
Figure 5.5 Similar bug search and result list	41
Figure 5.6 Developer feedback list for fixed issue.....	41
Figure 6.1 Experimental setup architecture for a software project.....	42

LIST OF TABLES

	Page
Table 4.1 Results before and after AAW	32
Table 5.1 Features of BugSTAiR Core Services.....	35
Table 5.2 Features of BugSTAiR Core Services.....	36
Table 6.1 Dataset Statistics.....	43
Table 6.2 Comparison of IRBL tools	44
Table 6.3 Comparison of experimental results.....	46
Table 6.4 Execution Time Statistics of BugSTAiR.....	48



CHAPTER 1

INTRODUCTION

1.1 General

Many studies have been conducted to reduce maintenance costs in software development processes and to improve the quality of software, considering different metrics. The typical Software Development Life Cycle (SDLC) process consists of iterative phases ranging from requirements analysis to maintenance. There can be various issues in each step that threaten the quality of the software. Software bugs are one of the most critical threats in this process since they are visible to the end-user and reduce customer's confidence in the software. The maintenance phase of the SDLC starts after the release of the software, and its cost is generally more than development costs for large scale software projects. For larger software projects, catching and fixing implementation errors becomes more difficult. Therefore, it is important to find a buggy source to reduce maintenance time and cost. Bug Localization (BL) is one of the ways in which developers use bug reports from bug tracking systems. The bug tracking system is a part of the issue tracking, which is dedicated to the software development process. All stakeholders, such as developers and quality assurance engineers, use these tools to track progress on bug fixing. Then, they have to overcome the time-consuming challenges such as reproducing the bug as specified in the bug report, understanding the coding structure, programming logic, and goal of the related flow, etc. For this reason, there should be some efficient methods to automate BL according to the bug reports.

In general, there are two different methods used in BL. One of them is called dynamic BL, in which dynamic methods have some processes during the execution such as runtime traces, data monitoring, tracking execution flows, etc. Researchers have developed many spectrum-based BL methods by inspecting parts of source code. Gopinath et al. implemented a different technique that combines spectrum-based BL with specification-based analysis to overcome the spectrum-based BL method's limitations. The other approach is static BL, which uses bug reports and source code to locate bugs. Static BL methods are easy to apply on any phase of the SDLC since

they have few external dependencies and relatively low computational costs owing to Information Retrieval (IR) algorithms. FindBug is a popular static BL tool that has been proposed by Hovemeyer and Pugh.

1.2 Purpose

The purpose of this research and thesis is to build a new bug localization algorithm that runs on different datasets that are implemented with various software development languages.

Both the previous methods and implementation of approaches are evaluated on well-known open-source datasets like Eclipse, AspectJ, SWT. All of them are developed with the Java programming language. Some implementation details of Java can help the IR process to have better accuracy. For example, a stack trace of an exception is a valuable input to indicate the buggy file and its function directly. In addition, the filename is always the same as the class name that is publicly declared. Unlike Java, JavaScript (JS) is very flexible and does not force any naming convention. Moreover, there is no experimental result on JS-based software and datasets in the literature. Another important issue is that development standards and implementation details may vary for the same programming languages depending on the company's coding standards. Besides, differences in project structures and language-specific keywords have forced us to understand the characteristics of a project. All the previous IRBL tools assign attribute weights intuitively or experimentally while retrieving data. For this reason, none of these tools can be a part of a commercial application or a service.

In short, there is a need for a software tool in order to run the proposed algorithm. BugSTAiR is a tool that this thesis also proposes, and it can be used as an on-premise application to track and support bug localization processes of a software company. BugSTAiR can integrate with project management systems to become a part of companies' SDLC.

1.3 Novel Contributions of this Thesis

This thesis has contributions on three levels;

First, we proposed a novel optimization algorithm, called Adaptive Attribute Weighting (AAW), to create an adaptive bug localization for every software project.

Second, we developed a bug localization tool, named BugSTAiR, which is the first tool that uses an adaptive weight calculation approach based on genetic algorithms. To the best of our knowledge, there is not any tool that works for software projects except well-known benchmark datasets. BugSTAiR makes this possible with the help of adaptation processes via the AAW algorithm. Therefore, any kind of software project can take advantage of BL.

Third, a new Eclipse dataset, which includes source code histories and bug reports of three major repositories, has been shared in open-source platforms for BL researchers. Eclipse project contains lots of different modules in different repositories. Therefore, any of the previous studies and works which use source code history did not use this dataset for their experiments. Our contribution will help future BL studies to compare their work to another benchmark dataset.

In short, in this thesis, (i) a novel adaptive attribute weighting algorithm, AAW, was developed, (ii) a novel BL tool named BugSTAiR is developed, (iii) a new dataset to BL study was introduced.

1.4 Organization of the Thesis

This thesis consists of seven chapters and the rest of this thesis is organized as follows.

Chapter 2 provides general information about relevant studies, literature review and field research on bug localization.

In Chapter 3, background information about bug localization processes, advantages, state-of-art algorithms, different approaches, and tools.

In Chapter 4, the new bug localization approach, AAW, and implementation details are explained.

In Chapter 5, the new bug localization tool, BugSTAiR, is detailly explained. The developed application is tested with different case study datasets.

In Chapter 6, many experiments were executed for the proposed bug localization approach with well-known and widely used benchmark datasets.

Finally, in Chapter 7, a summary of the thesis and proposed tool, conclusions, and future works are presented.

CHAPTER 2

RELATED WORK

In this chapter, literature, and field reviews are explained. Results of the research are discussed.

2.1 Literature Review

Along with the rapid development in the software industry, the importance of software maintainability and software quality has increased significantly. Software quality is evaluated with many different metrics. Software development lifecycle has iterative phases from requirements analysis to maintenance. There can be various issues in each phase that threatens the quality of software. Software bugs are one of the most important threats to this process since they are visible and reduce customer's confidence in the software.

The maintenance phase of the software development lifecycle starts after the software is released, and sometimes maintenance cost can be more than development phases in large scale software projects. When the scale of the software project getting bigger, it is difficult to find and fix coding/implementation errors. Therefore, it is important to find the buggy source to reduce maintenance time and cost.

There are some steps established during the bug fixing processes. The first step is bug localization. In this step, developers use a bug report/information from the bug/issue tracking system. After they have to overcome the time-consuming challenges such as reproducing the bug according to the bug report, understanding the coding structure, programming logic and goal of the related flow, etc. Therefore, there should be some efficient methods to automate bug localization according to the bug reports.

In general, there are two different approaches in bug localization. One of them is

dynamic bug localization. Dynamic methods have some processes on the applications' execution time, such as runtime traces, data monitoring, tracking execution flows, etc. Researchers have proposed various spectrum-based bug localization methods by inspecting a small part of source code. Gopinath et al. proposed a technique that combines spectrum-based bug localization with specification-based analysis to overcome the spectrum-based bug localization method's limitations. The other approach is static bug localization. Static methods use bug reports and source code for analysis processes to locate buggy sources.

Static bug localization methods are easy to apply on any phase of software development since they have few external dependencies and have relatively low computational cost with the help of Information Retrieval (IR) algorithms. FindBug is a popular static bug localization tool that has been proposed by Hovemeyer and Pugh (2004).

Recently, many researchers have worked on IR-based bug localization techniques. IR is the science that deals with the representation, storage, organization of and access to information items. IR approaches have two important concepts, such as query and document collection. Each bug report represents a query and the source files to be searched indicate the document collection. IR techniques use these inputs to rank documents by relevance, according to the ranked list of candidate source files that may contain the bug. The ranking process has consecutive phases that start with bug report creation. Then the user enters a bug report query into the system then IR techniques compute a numeric score for all candidate source files that match the bug query. Finally, top-ranking candidate source files are listed for developer consideration.

The success of an IR technique is highly dependent on algorithms used in retrieval processes. Rao et al. have compared the main IR techniques; Unigram, Vector Space Model (VSM), Cluster-Based, Latent Dirichlet Allocation, Latent Semantic Analysis (LSA), and some various combinations.

Poshyvanyk et al. have developed PROMESIR, which uses a probabilistic ranking method and a data acquisition method called Latent Semantic Indexing (LSI). Ngyuen et al. customized the LDA approach and proposed BugScout. Therefore, different approaches and techniques are proposed to improve the efficiency of information retrieval and bug localization processes. According to these improvements, BugScout took the lead because of its good performance on some large-scale datasets because most of the researchers used a few bugs in the evaluation process before BugScout.

Zhou et al. proposed BugLocator that used rVSM(revised Vector Space Model) and performed on some large-scale open-source projects. BugLocator uses text similarity between source files and bug reports. Also, it uses information about previously fixed bugs to improve bug localization accuracy.

BugLocator has better experimental results than BugScout on compared datasets. Then, another approach is introduced by Saha et al. BLUIR (Bug Localization Using Information Retrieval) uses structured information analysis of source code such as class names, method names, etc. BLUIR has located more bugs than BugLocator, according to the experimental results on the same datasets. Thus, using structured information of a source file is more efficient than a simple source file as a document.

Youm et al. has proposed BLIA (Bug Localization using Integrated Analysis) by using some other information about bugs. In addition to the bug similarity information that BugLocator used, BLIA uses stack traces, comments in the bug report, and change the history of the source code to have better accuracy. Also, BLIA is a bug localization algorithm that provides multi-level scoring such as method-level scoring, file-level scoring, etc. In order to compare, BLIA was evaluated on the same datasets, which was used to evaluate for both BugLocator and BLUIR. According to the evaluation results, BLIA was better than both BLUIR and BugLocator in those datasets.

All these approaches are evaluated on well-known open-source datasets. These datasets are Eclipse, AspectJ, SWT and etc. All of these software's are developed with Java programming language. Some of the implementation details of Java programming language can help to information retrieval process to have better accuracy.

2.2 Field Review

There are some academic tools that have been developed for bug localization techniques. All of these tools are used for only academic purposes. There is not any commercial tool for software bug localization due to the diversity of software development languages, design patterns, and coding standards that are used in different companies. Therefore, all bug localization tools and researches use some well-known datasets and compare their experimental results with each other. Table 2.1 presents the comparison of bug localization tools and key contributions of their approaches.

The proposed tool, BugSTAiR, uses Adaptive Attribute Weighting (AAW) process to adopt a new software project for bug localization process. BugSTAiR can optimize the retrieval results by generating project-specific coefficients, which are used in the query to retrieve. The differences between bug localization tools shown in Table 2.1 are used algorithms and data source while dealing with the source code.

Table 2.1 Comparison of bug localization tools

Approach	BLUIR	BRTracer	BLIA	BugLocator	BugSTAiR
Published	2013	2012	2015	2012	2020
IR Method	TF.IDF	rVSM	rVSM	rVSM	Lucene
Structured Information	O	X	O	X	O
Bug Similarity	O	O	O	O	O
Version History	X	X	O	X	O
Stack Trace Analysis	X	O	O	X	X
Adaptive Attribute Weighting	X	X	X	X	O

CHAPTER 3

BACKGROUND INFORMATION

Software engineering is an engineering discipline that applies software development processes to improve development efficiency, quality assurance, and provide better project management. Software development lifecycle (SDLC) has several steps to ensure all these improvements. This thesis focuses mostly on the development and maintenance phases of SDLC. Bugs can occur both in the development and maintenance phases. Therefore, a bug can be caught by quality assurance engineers or end-users, and they report the bug. A customer reported bug should be fixed as fast as possible to provide customer satisfaction. Catching and fixing the bug is very important. Therefore, automatically detecting and locating bugs in software projects can improve to a great extent the software quality as it eases the effort in fixing bugs and increases the efficiency of quality assurance. Useful data in achieving this can be found in source code comments and bug report tickets but are mostly overlooked in some of the existing methodologies.

In this chapter, we are going to explain and discuss background information about some of the recent popular methodologies proposed for bug localization.

3.1 Bug Localization

Bug localization is a process to find a bug in software source code according to the given bug report. Bug reports provided by a customer or a test engineer have much information about the bug, such as; necessary information about flow, steps to reproduce, given inputs, etc. There are several approaches that deal with bug localization. These approaches are information retrieval-based bug localization (IRBL), neural network-based bug localization (via convolutional neural network or deep learning), and some experimental studies on hybrid solutions. In this chapter, brief information about some well-known and performing tools on each approach is given.

3.2 Deep Learning-Based Approaches to Bug Localization

Manually locating bugs in large software projects could introduce a great deal of time and effort for software developers. In recent history, there have been a number of significant state-of-the-art approaches proposed for bug localization that would aid software developers in the process of bug fixing. On the contrary, practitioners have been in search of such a model that could satisfy their requirements in terms of reliability, scalability, and efficiency. Most of the currently proposed approaches hardly meet these requirements and causes practitioners to steer away from adopting them in their software development life cycles. However, some recent advances in deep learning-based techniques have proven to outperform existing state-of-the-art models in bug localization.

Polisetty et al. have composed a CNN and a Simple Logistic model to experiment and compare the efficiency of deep learning-based models to state-of-the-art models used in bug localization. They have trained their deep-learning based models with a number of five different open-source software projects, all of which are written in Java, and compared their performance to some of the state-of-the-art models trained on the same datasets.

Their work has shown higher performance results when compared to conventional machine learning-based state-of-the-art approaches. However, they have proven that deep learning-based methodologies are still yet to fully satisfy the requirements of practitioners. Their work also has proven a need for a standardized performance benchmark among bug localization methodologies for a realistic and fair evaluation.

DeepLoc is one of the well-known bug localization approaches listed in Polisetty et al.'s comparison. DeepLoc is a project that has been conceived in recent years, has been being experimented on software projects like AspectJ, Eclipse, JDT, SWT and Tomcat. It consists of an enhanced convolutional neural network (CNN). This network takes into account the recent bug fixes and their frequency and utilizes word-

embeddings and feature-detecting to compose a prediction. A set of purpose-specific CNNs use these bug reports and the source code semantics represented in the word-embeddings to detect patterns features in them. Figure 3.1 below depicts the overall architecture of DeepLoc, revealing the offline training stage as well as how a trained model predicts the faulty source files from new bug reports.

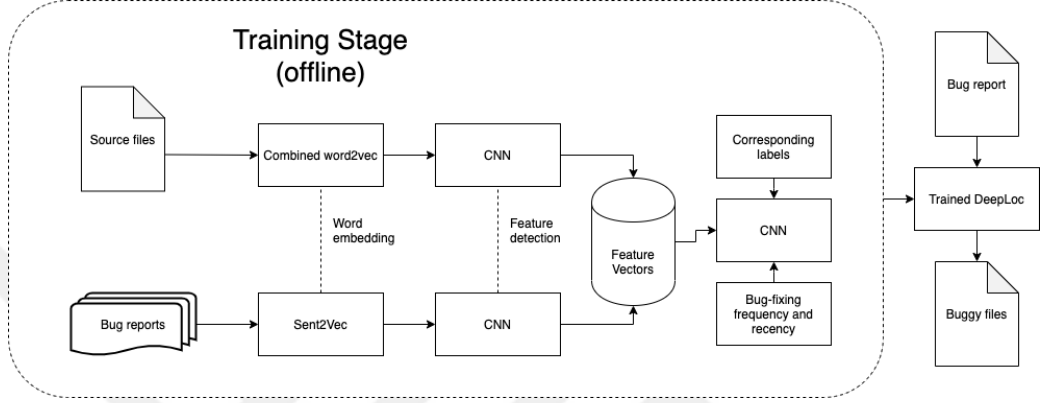


Figure 3.1 The overall architecture of DeepLoc

The results of experiments have proven that DeepLoc achieves a higher MAP (mean average precision) score by 10.87%–13.4% than conventional CNNs. It also outperforms four of the state-of-the-art approaches in bug localization (DeepLocator, HyLoc, LR+WE, and BugLocator) in terms of Accuracy@k, MAP, and MRR (mean reciprocal rank) scores. Accuracy@k stands for the bug reports percent in which at least an actual file with a bug located in it is placed somewhere within the top-k rank.

3.3 Information-Based Approaches to Bug Localization

In the literature, there are many open-source software products with datasets that include lots of bug summaries. However, it is difficult to find web applications that are developed by JS-based frameworks having a bug report dataset. Figure 3.2 illustrates a real-world bug report from a commercial application developed for a bank.

IRBL approaches rely on calculating similarity scores between bug reports and source code files according to the results of similarity matching algorithms. All source code files have a computed similarity score for each bug report. BL has some steps which should be executed in a predefined sequence. Detailed information about these steps is given in Chapter 4.

Bug ID: 1433 Summary: Account Settings – Next Button is not working after entering security question . Button should trigger a redirect action to the success state
Source Code File: securityQuestionPageHelperFactory.js
<pre> angular.module("WebApp.core").factory("securityQuestionPageHelperFactory", function (securityQuestionApi, smsOtpConfigFactory, smsOtp, \$state) { "use strict"; function securityQuestionPageHelper(securityQuestionPage) { var self = this; this.validateSecurityQuestionAnswer = function () { securityQuestionApi.validateSecurityQuestion({ "answer": securityQuestionPage.formData.answer }, { "skipDefaultErrorAlert": false, "onSuccess": self.onValidateSecurityQuestionSuccess }); }; this.onValidateSecurityQuestionSuccess = function (resp) { if (securityQuestionPage.config.smsOtp) { self.startSmsOtpFlow(); } else { \$state.go(securityQuestionPage.config.successState.name, securityQuestionPage.config.successState.params, {"location": "replace"}); } }; } }; </pre>

Figure 3.2 Bug summary of a real-world application

3.3.1 Common Bug Localization Process

IRBL approaches have five main steps, as shown in Figure 3.3. They are pre-processing, indexing, query construction, similarity computation, and retrieval.

- **Pre-processing:** This step is related to both source code files and bug reports. All of them should be pre-processed to improve the efficiency of the retrieval process. In this step, all stop-words such as language-specific identifiers and punctuations are removed from the source code. In addition, some syntactic operations are performed, such as camel case splitting, lowercase transformation, word stemming, and tokenization.
- **Indexing:** IRBL approaches are used to index a dataset that is ready when both source file and bug reports are pre-processed and the dataset is prepared. The VSM is one of the well-known IR techniques, but there are also some other

probabilistic models such as LSI. The Eclipse dataset includes source code histories and bug reports.

- **Query Construction:** Query is one of the most important parts in IR processes. In general, summary and description fields of bug reports are used as input.
- **Similarity Computation:** There are several methods to compute the similarity between bug reports and source code files. Every IRBL approach applies one of these methods to compute relevance.
- **Retrieval:** After all steps are performed, each IRBL approach applies its proposed algorithm or method to obtain better accuracy on the retrieval process.

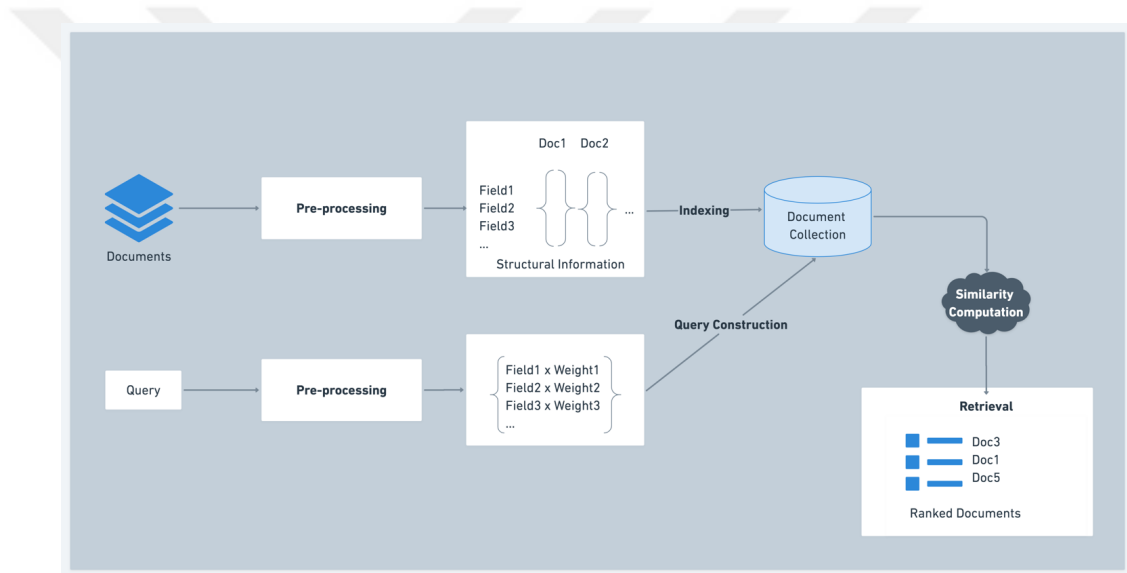


Figure 3.3 General view of IRBL processes

General information about IRBL processes and steps are given. The well-known IRBL tools such as BugLocator, BLUIR and BLIA and their approaches are discussed.

3.3.2 *BugLocator*

BugLocator is an Information Retrieval (IR) based approach to bug localization. It predicts the relevant files that would potentially help to fix a software bug.

BugLocator utilizes a revised Vector Space Model (rVSM) to rank all the source code files in terms of the textual similarity between the initially reported description of the bug and the source code itself while using the previous information on similar bug reports that have been resolved as well. In Figure 3.4, the overview of BugLocator’s structure is shown.

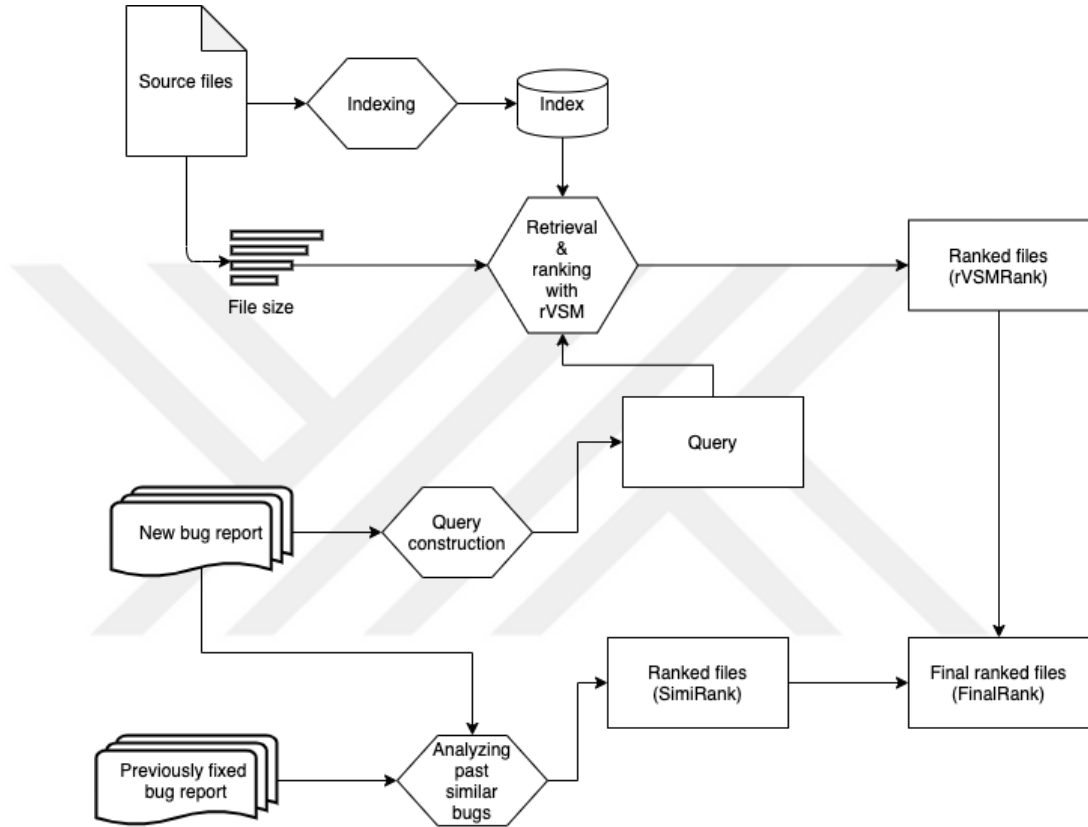


Figure 3.4 The overall architecture of BugLocator

The model has been experimented on a number of four different large-scale open-source software projects to predict the location of more than 3,000 bugs. The experiment results have proven that BugLocator can predict the source code files that are potentially contaminated with software bugs with high accuracy. For the software project Eclipse 3.1, 62.60% of the estimated buggy files are ranked in the top ten among 12,863 files. The experiments have shown that BugLocator outperforms the existing state-of-the-art bug localization models with high accuracy.

3.3.4 BLUIR

BLUIR is another approach in bug localization that utilized information retrieval techniques. While these information retrieve-based bug localization systems are highly scalable for larger software projects, their performance inaccurately narrowing down their prediction to a small number of potential source files remain relatively low in localizing bugs.

The novelty of BLUIR is its captured insight that the use of structural information retrieved from the code constructs, namely, class names and method names, allows for a more accurate and highly performant prediction of bug localization. As previously mentioned, approaches BugLocator and DeepLoc, Bluir makes use of bug report descriptions along with the actual source code to conducts its prediction. It also takes advantage of previous bug similarities when applicable to further fine-tune the localization. Figure 3.5 is the overall architecture of BLUIR.

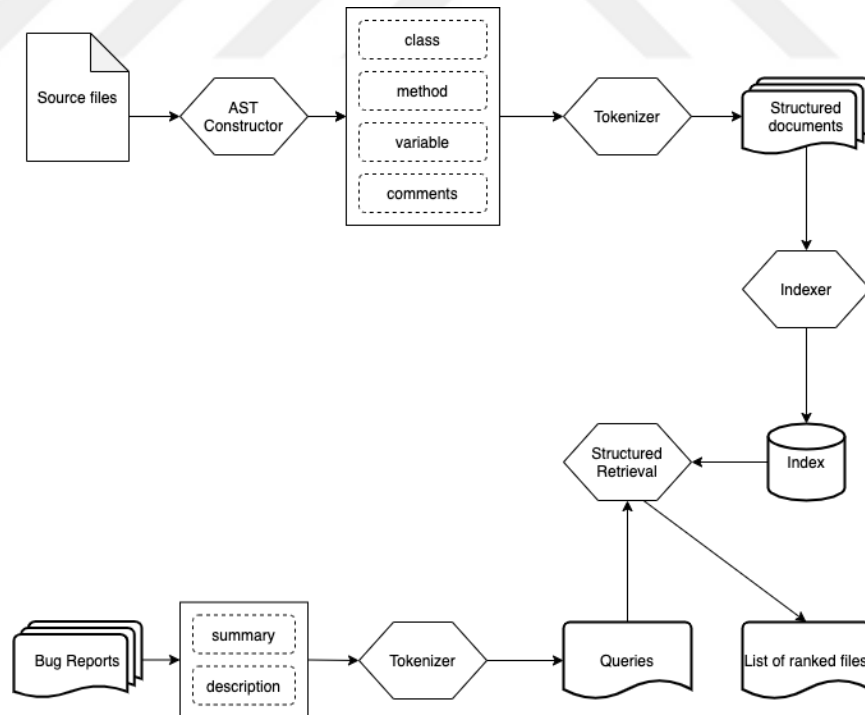


Figure 3.5 The overall architecture of BLUIR

The approach constructed in BLUIR provides a ground basis for information retrieval-based approaches in fundamental bug localization researches in terms of both

theoretical and empirical knowledge. BLUIR is evaluated on a number of four different large-scale open-source software projects on a number of approximately 3,400 bugs. The results have proved that BLUIR matches and even outperforms existing state-of-the-art models even for the cases where BLUIR does not utilize the bug similarity data used in other models.

3.3.5 BLIA

BLIA is another information retrieval-based bug localization model that aims to increase accuracy in predicting faulty source files by incorporating in bug reports, structured data from source files, and the source code change history.

BLIA introduces a novel approach to information retrieval-based models, a so-called *Integrated Analysis*, wherein a plethora of structured and unstructured data is aggregated to form a more accurate prediction of potential faults in source files. To do so, BLIA utilizes textual descriptions and comments that are fetched from bug reports while analyzing the stack trace of the program, the structured information gathered from the source files, and finally, the source code change history. This approach helps BLIA to narrow its localization of the bugs from a file-level down to a function scope level using data from the previous bug repository. It can be seen the workflow BLIA employs from Figure 3.6.

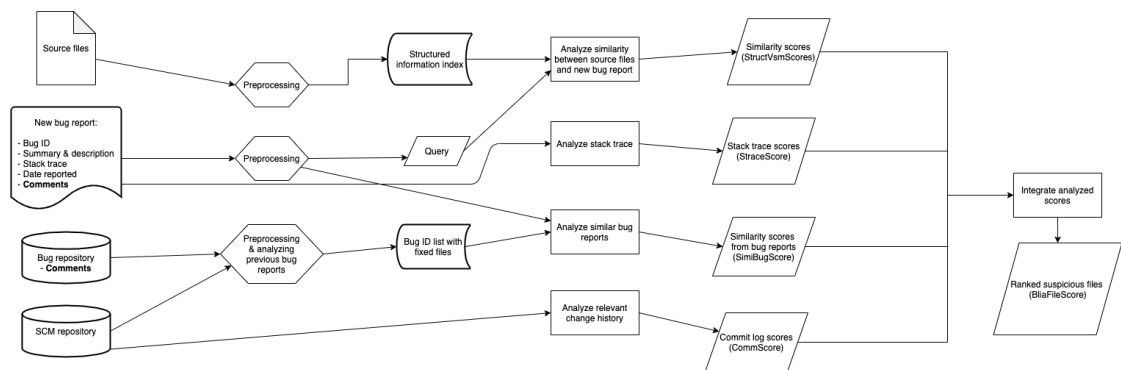


Figure 3.6 BLIA's workflow

BLIA has been evaluated on a number of three large-scale open-source software projects, namely, AspectJ, SWT, and ZXing. BLIA surpassed some of the state-of-the-art approaches in bug localization in terms of mean average precision (MAP), such as BugLocator, Blur, and BRTracer by 54%, 42%, and 30%, respectively.

The new approach proposed in BLIA not only outperforms its existing counterparts but also allowed for an improved granularity level of bug localization, from the file level to a function level.



CHAPTER 4

ADAPTIVE ATTRIBUTE WEIGHTING ALGORITHM

In this thesis, we propose a new approach to widen bug localization process to all software projects. Bug localization is a process to discover information from source codes and bug reports. There are several types of research on bug localization area. Although all of the studies are useful academic researches, they do not offer a solution for use in production. Therefore, the software industry needs a bug localization tool that is capable of adapting any software project and able to be successful in localizing new bugs. The aim of the proposed algorithm, titled Adaptive Attribute Weighting (AAW), is to adapt and prepare a software project for bug localization, an adaptation process is constructed based on genetic algorithms. Therefore, we defined an adaptation process that actually lies on the AAW algorithm to prepare the basis of a software project to bug localization. In this chapter, detailed information and implementation of the adaptation process and AAW are given.

4.1 Adaptation Process

Software projects can be developed with different programming languages such as Java, Javascript, C++, Swift, Kotlin, etc. Therefore, there is not only one language while dealing with bug localization as previous researches did. Every language has its own structure and standards to follow. Moreover, every software company has internal software development standards such as coding notations, design patterns, styles, etc. The adaptation process provides us to handle this diversity in the bug localization process.

Adaptation processes have iterative steps to prepare newcomer projects to bug localization such as pre-processing, indexing, query construction, retrieval, and optimization. The first four steps of adaptation are also basics steps of IR-based bug localization. In this chapter, the proposed adaptation process and AAW algorithm are presented.

4.1.1 IRBL Processes in BugSTAiR

All of the previous researchers that are pointed out in Chapter 2 have studied on software projects implemented with Java. According to this fact, all benchmark datasets have Java-specific information. The proposed model has generic processes because Java is not the only programming language in real life. JS is the leader among the list of most popular programming, scripting, and markup languages. Therefore, this study focuses on the applications of not only Java-based but also JS-based to build a generic retrieval model. A new “Adaptation” step is defined to provide this generic architecture. The aim of this step is to build IRBL basis for newcomer software and optimizing the retrieval process.

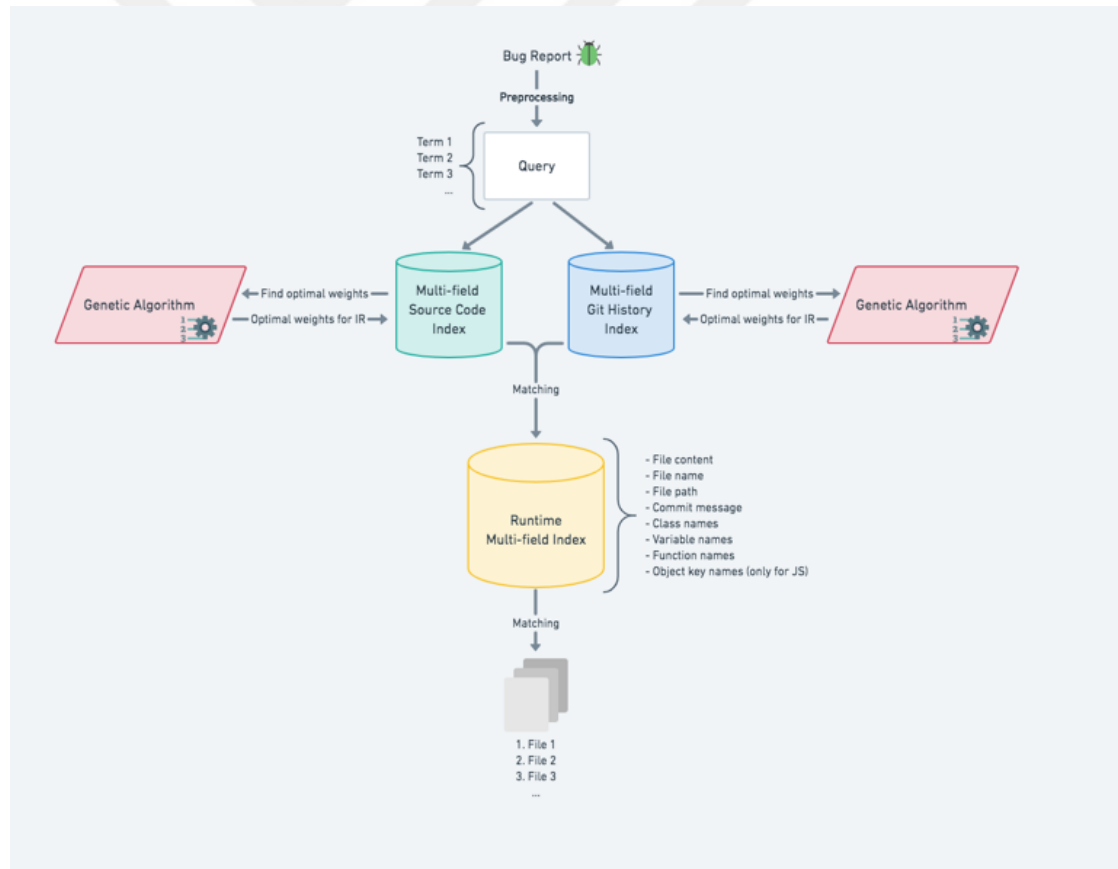


Figure 4.1 General view of BugSTAiR architecture

IRBL in JS-based web applications is hard to implement, and retrieval results are not as accurate as in Java-based software applications. The main reason is that the user interface (UI) of an application is related with more than one file at the same time. Furthermore, a web application may have many files with the same name but with different file extensions such as featurex.html, featurex.css and featurex.js. This situation causes extra complexity for all computations in the process. Thus, the proposed approach focuses on locating non-UI related bugs such as logic and flow in web applications. Therefore, BugSTAiR evaluates only source files with “.js” extension while working on JS-based applications. The other UI-related project files such as “HTML” and “CSS” are out of the scope for this study. Figure 4.1 shows the general architecture of the proposed approach.

In our proposed work, it is considered that the change history of the source code is as important as current source code because any change in the source code has a history. This history can be related to a feature or a bug fix. There may be many source code files depending on the change. Evaluating the information obtained from the history, the impact analysis between the features and source code has been identified. Therefore, the history of source code is valuable and is used to locate the potential buggy file. The steps of the proposed approach are given in Figure 4.2.

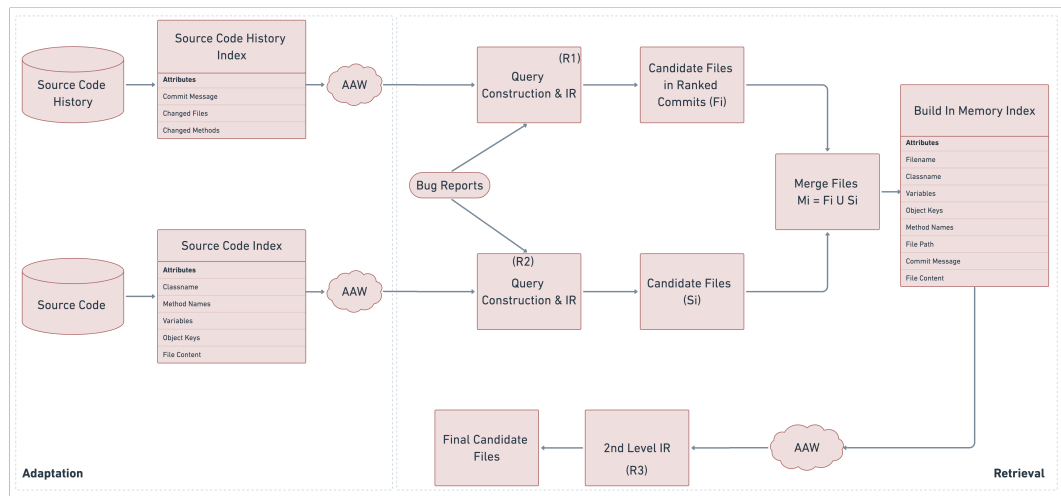


Figure 4.2 Steps of BugSTAiR

In Chapter 3, common bug localization steps are explained. In this section, the implementation details of the proposed approach's steps are considered. The adaptation step starts with pre-processing. In this step, different stop-words lists are defined for different programming languages. Then, different source code parsers have been used to understand the source code structure. The same stop-word list and language parser are used with BLIA when processing on Java-based applications.

However, the new stop-word list and JS parser have been created to process JS-based applications. The stop-word list contains a list of words that are commonly found in languages that carry very little or no significant semantic context in a sentence. The JS-based stop-word list contains natural language words that are already in Java-based stop-word lists and also JS language-specific keywords. Details of pre-processing are covered in Chapter 3.2. The adaptation step includes three AAW execution to get ready for retrieval. The first two execution run after the pre-processing step in which two different indexes have been built, such as source code index and change history index. The structures of these indexes are different. Source code index has five attributes, which are: class names, methods, variables, file content, and object keys. Change history index has three attributes, which are commit messages, changed files, and changed methods.

The proposed work has been built on a service-oriented architecture. Each web service can run asynchronously in different instances. All necessary services are also packaged in a container to scale if needed. Therefore, these two indexes could be created simultaneously. At this point, AAW processes run for both indexes to identify the best weights for retrieval. Next, the retrieval query is built as the query construction algorithm given in Section 3.5. Then, the first retrieval is performed based on the change history index with AAW on all attributes. Also, another retrieval is performed based on the source code index. Apache Lucene provides us similarity scores between commit messages and bug reports. Every commit may have files that are changed more than once. Therefore, scores of the files that are found in the retrieval processes have to be consolidated. Finally, a buggy file list is gathered according to the first level of

the IR process output. Then, the second level of the IR processes starts by creating a structural in-memory index including files in the consolidated list. The proposed structure includes class names, method names, variables, object keys and file content from the source code index. In addition, filename, file path, and commit messages are included from the source code history index as an attribute to the in-memory index to combine all information in one index. Then, the optimum attribute weights are calculated according to the output of the genetic algorithm in the third execution of AAW. File scores that come from the third execution of AAW are used for re-ranking between candidate files coming from the first level IR. After the re-ranking process, a final score is generated for each file. So, they can be used for any application that helps software development teams to find bugs earlier in the maintenance period of the SDLC.

The idea behind including GA in the approach is to reduce the impact of changing project standards and application development standards on the model and to achieve more precise results.

The multi-dimensional search on different fields, and combining these results conducted using IR techniques requires a dynamic calculation of the coefficients /weights that affect the search process and retrieval results. Another approach to find and use weights to produce the best results is the Brute Force Search algorithm. As the size of the dataset grows, it is better to use optimization methods since the time for calculation is high, and this has to be repeated in specific periods (new records, daily, etc.).

4.1.2 Source Code – Bug Report Pre-processing

During the BL process, HTML and CSS files are excluded from the source code repository, and JS files are the only accepted input to be processed. Moreover, all UI-related bugs are eliminated while getting bug reports from the issue tracking system. Then, all stop-words are removed from the source code files and bug reports. Stop-

word removal is most common process to reduce dimensionality. (Kılınç, 2019) Stop-words include keywords such as;

- English stop-words: “a”, “the”, “to” etc.
- Syntactic symbols/identifiers: “null”, “undefined”, “alert”, “init” etc.
- Operators / Punctuations: “==”, “!=”, “<”, “>” etc.

There are some different naming conventions in software companies. Identifiers may consist of more than one word. In order to increase the accuracy of the retrieval, identifiers are tokenized. To achieve this, individual tokens are used, but there might be some conflicts between bug reports and source code with regard to case sensitivity. To resolve these conflicts, all texts are transformed to lowercase.

In addition, to be the first study on the web front-end side, the model is tested in Java-based systems so that the pre-processing step can be done similarly. A language-based stop-word list has also been created to build a generic infrastructure. The language parsers are used to understand the written code structurally. JS parser and Java parser are included in the proposed approach. To support different software development languages, the language-based stop-word list is included in the collection, and a language parser is added to the project to understand the structure of the new language.

4.1.3 Indexing

In this section, the source code indexing approach is presented. Lucene (Apache, 2020) is used to index the source code files. Lucene is one of the most widely used and well-known open-source IR systems. In the proposed approach, Lucene evaluates a relevance score between source code files and bug reports. There are two ways for multi-parameter indexing. Some researchers prefer building a different index for each attribute, and the others use one index including all attributes. In this study, building an index with all attributes is preferred with Lucene’s powerful APIs.

Before the indexing step, the source codes have to be parsed to index them in a structured way. After the source codes are parsed, important information such as class, method, variable and function names are extracted. All source code files are analyzed structurally by using specific language parsers. Each source code file is called “document” in the index structure and any valuable parts of the source code are called “attributes”. These attributes are added to the document as a field. Both filenames and attribute names are usually written in camelCase naming convention. In this way, all the filenames and attribute names which have more than one word are added to the related field of the document by using the camelCase notation. It is also discovered that these names may be included in the bug records and they are also indexed as a separate field in the document to increase success. For example, “securityQuestionsPageHelper” method name is added to the method field of the index with five inputs which are security, question, page, helper, “securityQuestionPageHelper”. In addition to these structural fields in the document, unprocessed and flat text of the source code file is stored in the document.

First, the bug report is searched for commit messages in the source code history index. In this way, it is aimed to find the most similar solution set from the previous commitments to the solution of this error. The source code files obtained from this set of solutions are indexed by the method described above. After these indexes are formed, the process of finding the weights of the attributes begins. By means of the genetic algorithm, the optimum values of the attributes on both indexes are determined to be used in the next step. The obtained values help in-memory index to achieve the best result on the entire data set.

4.1.4 Query Construction

Source code files are called as document collection and bug reports are evaluated as query in IRBL process. Since the bug reports are pre-processed in the first step, query construction is performed in the retrieval. According to the previous studies, it is understood that the query construction process is very important and critical for retrieval accuracy. Many researchers use special weights for fields on documents while

constructing the retrieval query according to their empirical studies on each dataset. In our approach, outputs of the AAW process are used to set weights while constructing the retrieval queries. In addition, all words of bug report summaries that are tokenized and pre-processed in the previous steps are used in the queries. In general, bug summary contains useful information to localize bugs. Moreover, the description parts of the bug reports are examined to verify whether they have valuable information about bug or not. Then, it is decided to include bug descriptions to queries. Bag of words algorithm (BoW) is applied to extract more valuable information from the bug description and to reduce complexity of query construction. Finally, ten words are selected according to the word counts to be added to the query.

Apache Lucene is used for indexing, so the Lucene APIs are used in the retrieval phase. Lucene API provides some methods to retrieve data according to the query that has multiple attributes with different coefficients. Figure 4.3 shows query construction approach.

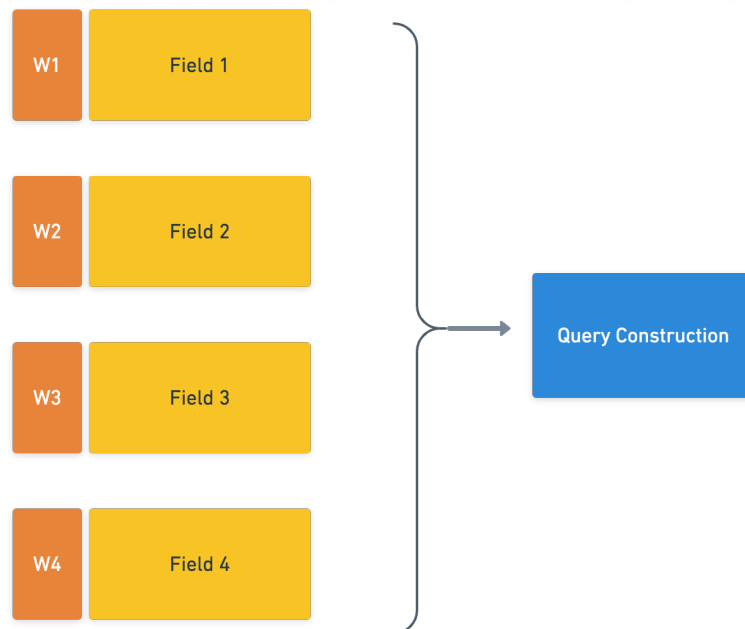


Figure 4.3 Query construction approach

4.1.5 In-memory Source Index (IMSI)

IMSI has an important role in the project. As a result of retrieval on source code history, the most similar change set is determined according to the similarity between commit messages and bug reports. Every commit may include more than one file. Therefore, unique source files are created by evaluating the changes. On the other hand, the code pieces that may potentially contain bugs are determined as a result of the IR process on structured information of source codes.

The merge operation in adaptation process is executed simultaneously. The candidate bug resources selected after the IR process on source file are merged with the candidate file names according to the IR results on the source code history. All files in the merged list are re-indexed in a more complex structured information during the execution of the application. It is thought that indexing is a time-consuming process. Therefore, lists of candidate source files to index are filtered. Through, building an IMSI does not affect the retrieval process dramatically.

After IMSI is constructed, there are more attributes for query construction in IRBL process. New weights for each attribute are required. By executing the AAW process on the attributes, new weights are determined for all attributes. So all prerequisites get ready for the second level of IR.

4.1.6 Adaptive Attribute Weighting

GA is a widely used search and optimization method that works in a manner similar to the evolutionary process observed in nature. It seeks the best holistic solution based on the principle of survival in complex multi-dimensional search space. GA has three main steps: crossover, mutation, and selection.

In this section, all configurations and strategies which are chosen for implementation explained in detail. Initially, each chromosome is designed to have eight genes which can take double values (0.0 – 1.0). Each gene is represented by 16-bits and can have fractions up to two decimal points. A set of chromosomes are defined as population, also population is subset of a solutions in current generation. There are some limitations while defining the size of population. The diversity of the population should be maintained otherwise it might lead to premature convergence. The population size should not be kept very large as it can cause to GA to slow down, while a smaller population might not be enough for a good crossing pool. As mentioned before, diversity of the population effects optimality and initial population is important. Random initial populations which is increasing the diversity of the chromosomes in the population. In this approach, the initial population is generated completely random with minimum sample size of 100 and maximum sample size of 200. General view of GA flow is shown in Figure 4.4.

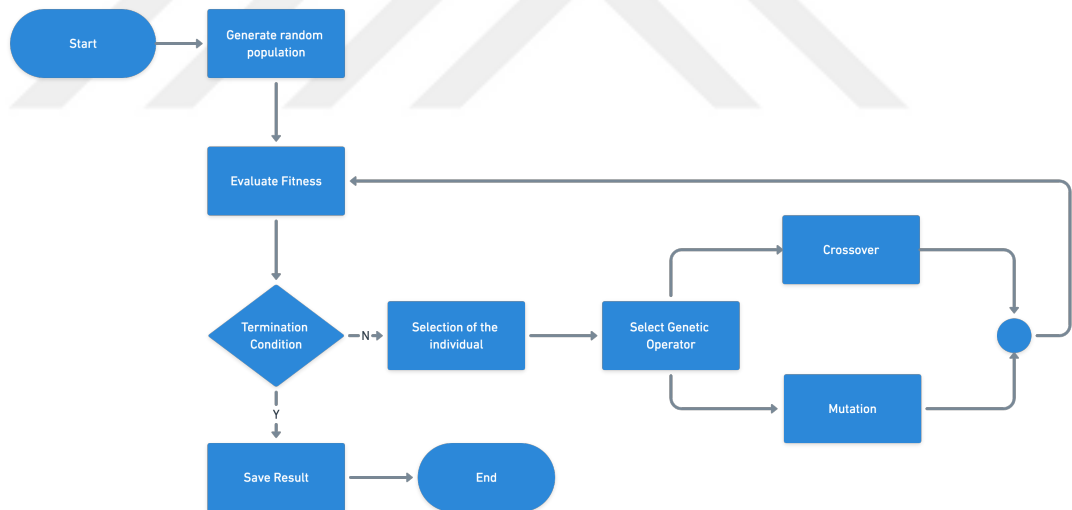


Figure 4.4 GA flow

After a brief information about population which is used in GA, crossover strategy and rate are implemented in the proposed approach. Crossover step is similar to reproduction and biological crossover. More than one parent is selected and one or more off-springs are produced using the genetic material of parents. Crossover has two internal steps as following;

- Sort the parents according to fitness scores from large to small
- Divide them into groups of two and cross them among themselves to generate new off-springs

Uniform crossover strategy is used to generate new off-springs with the mix probability of 0.75. In uniform crossover, each gene is evaluated separately while deciding it to be included in the off-springs.

Second important process that provides diversity is mutation. Flip Bit Mutation (FBM) strategy is used in the proposed approach. FBM is a mutation approach which has two steps such as selecting one or more bits randomly and flip them with a given probability. In this approach, 0.1 is given as mutation probability. As mentioned before, crossover and mutation are two critical methods to create a new chromosome using existing genes. The selection of the genes which occurs with respect to crossover and mutation is called the selection process. Consequently, complex problems are solved to inform the GA of which gene is good using a fitness function and coding variables. The steps of GA can be summarized as follows:

- Generate the initial population randomly
- Find the fitness score for each chromosome in the population
- Perform gene reproduction using crossover and mutation operations.
- Eliminate chromosomes having inadequate fitness score.
- Repeat Step 3 through 5. The process is terminated after the configured number of chromosomes are generated.

In this approach, GA is used to find a common solution for all datasets to solve attribute weighting problems. GA must have a fitness function to optimize the given problem. Therefore, a customized fitness function has been implemented to solve the current problem. This function is problem dependent and each problem has its own fitness function. For example, error measures such as Euclidian, Manhattan have been

widely adopted in classification problems. On the other hand, entropy-based approaches can be used to solve different problems. Details of proposed fitness function are explained. The Gene/Population selection algorithm is the most important part in GA. Elite selection (elitism) is the most successful and preferred method in the literature. Especially, elitism strategy has been widely used in different evolutionary algorithms. Therefore, the Elite selection algorithm is used for the selection process. This selection method is optimized to choose the best chromosomes. The custom fitness function executes the required number of multiples and keeps the result in memory. After the required number of chromosomes are generated (min 100, max 200 is used in the application), the existing population is evaluated. The evaluation step is done considering the termination function. This function decides whether the new generation should be produced or not. The application is terminated if the best chromosome value is not changed after the configured number of iterations are executed. Otherwise, new chromosomes and populations continue to be produced by the GA iteratively.

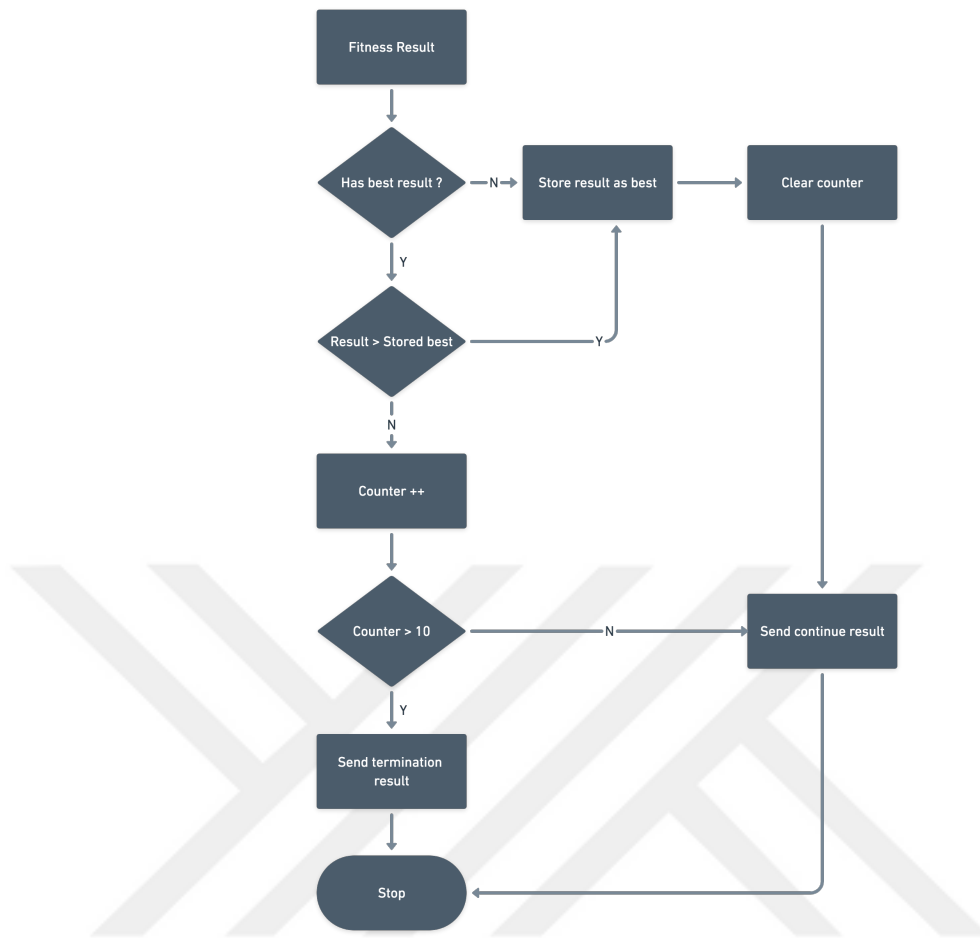


Figure 4.5 Termination condition of GA

In this study, GA is used to optimize IR efficiency by finding the best coefficients for any given dataset. There are three AAW execution to optimize three IR processes. The main difference of these executions is the number of parameters which will be optimized. In this study, AAW algorithm is designed to support variable length parameters, and can take 3 to 8 parameters as input and returns optimized values for each parameter as output. Therefore, only one AAW implementation can be used in these three executions. Across to this, fitness function is different according to the index which is used for retrieval. AAW can switch a fitness function according to the number of input parameters. BugSTAiR performs the first execution of AAW on multi-field source code index. Therefore, fitness function evaluates the result for only this index. This index has five fields such as class names, methods, variables, file content and object keys. Fitness function evaluates every bug report in given dataset and compares the ranked files with Ground Truth Data (GTD). After all bug reports

are evaluated, fitness function returns Top 1 ratio of the IR which has coefficients from GA. The formula to calculate the base similarity score $R_1(j)$ is represented in the Equation 4.1.

$$R_1(j) = \sum_{i=1}^n (w1_{ij} \times q_i) + (w2_{ij} \times q_i) + \dots + (w5_{ij} \times q_i) \quad (4.1)$$

where, $R_1(j)$ is the similarity score of j^{th} source file, n is length of the vocabulary and, $w1_{ij}$ and q_i represents the weight of j^{th} class name and query of i^{th} bug report, respectively. $w2_{ij}$, $w3_{ij}$, $w4_{ij}$ and $w5_{ij}$ represents the weight of j^{th} methods, variables, file content and object key.

The second execution of AAW is performed on a multi-field source code history index. This index has only three fields such as commit messages, changed files and changed methods. Therefore, the fitness function in this execution can be represented in Equation 4.2.

$$R_2(j) = \sum_{i=1}^n (w1_{ij} \times q_i) + (w2_{ij} \times q_i) + (w3_{ij} \times q_i) \quad (4.2)$$

where, $R_2(j)$ is the similarity score of j^{th} commit message, n is the length of the vocabulary and, $w1_{ij}$, $w2_{ij}$, $w3_{ij}$ and q_i represents the weight of j^{th} commit message, changed files, changed methods and query of i^{th} bug report, respectively.

The third execution of AAW is performed on the in-memory index. Details of in-memory indexing are given in section 3.6. There are eight fields on this index and evaluation method is the same as previous executions. The formula to calculate the base similarity score $R_3(j)$ is represented in Equation 4.3.

$$R_3(j) = \sum_{i=1}^n (w1_{ij} \times q_i) + (w2_{ij} \times q_i) + \dots + (w8_{ij} \times q_i) \quad (4.3)$$

where, $R_3(j)$ is the similarity score of j^{th} source file, n is length of the vocabulary and q_i represents query i^{th} bug report. $w1_{ij}$ to $w8_{ij}$ are represents the weight of

related fields. The calculation algorithm is the same for all functions. The only difference among them is number of weights which used for each field included in query.

To summarize, all of the AAW executions are executed step by step including as fitness functions, selections, crossover and mutation configurations. All the parameters such as crossover, mutation probability and population size used in GA are selected with Grid Search (GS). It is a technique that scans the dataset to select optimal parameters for the constructed model. GS works iteratively on each data and compares the results for each value. Then, the best value for each parameter is found. The configurable parameters of the applied solution are mutation, crossover rate and the number of chromosomes. In order to understand whether the AAW algorithm performs better, basic IR tests have executed on benchmark datasets. The experiments are performed on four different datasets which are formed as initial structure on source code. The aim is to see better accuracies on benchmark datasets and apply the AAW to the IRBL model. After applying the AAW algorithm, retrieval results are better than retrieval results without AAW in the same conditions. The results of IRBL processes for each dataset are given in Table 4.1. AAW provides better accuracy on all four datasets that are used for benchmarking in bug localization. AAW achieves better accuracy compared to our previous experiments that have no specific coefficient/weight for each attribute. According to the accuracy results, improvements on benchmark projects such as SWT, Eclipse and AspectJ are 20%, 21%, and 62% respectively. These improvements on benchmark datasets show that the AAW improves the IRBL process results significantly.

Table 4.1 Results before and after AAW

Dataset	Before AAW (%)	After AAW (%)
SWT	47.00	56.32
Eclipse	26.19	31.61
AspectJ	20.84	33.80
Web Application	27.72	32.18



CHAPTER 5

A NEW BUG LOCALIZATION TOOL: BugSTAiR

Software quality is very important for project success. There are several projects that continue simultaneously in software companies. Each of them may be implemented in a different language but state-of-the-art bug localization tools do not provide a general or adaptive solution to handle this problem. Identifying the root cause of the bug and finding the possible buggy file is getting more time-consuming. Especially, larger projects are become more complex to manage and maintain according to the number of stakeholders in the project or size of the project. Therefore, industry needs a software tool in order to apply proposed adaptation process by AAW to real-word software projects. BugSTAiR is a tool which a software company can easily deploy and can integrate it to its SDLC. BugSTAiR also can be integrated to project management systems such as Atlassian Jira by the help of jira plugin. There are some other features that BugSTAiR have. All of them are discussed in this chapter.

5.1 BugSTAiR

The name of BugSTAiR is comes from “bug” and “stair”. The main idea is that developed tool should the first step of the bug localization processes. BugSTAiR has enough features to cover all parts of the processes.

BugSTAiR tool has three main component such as core services, ui dashboard and backend services. Details of each component is given in this chapter.

5.1.1 *BugSTAiR Core Services*

BugSTAiR Core services module can include features about adaptation processes. We prepared a statement-of-work (SoW) document to identify all the needs. Feature specifications is given in Table 5.1.

Table 5.1 Features of BugSTAiR Core Services

No	Feature	Epic	Description
1	Operations	Indexing	<p>The system works on independent services.</p> <p>There must be two kind of indexing mechanism.</p> <ul style="list-style-type: none"> • Source Code Indexing • Source Code History Indexing
2	Operations	Configurations	<p>User can manually configure the BugSTAiR parameters.</p> <p>Parameters can be;</p> <ul style="list-style-type: none"> • Source Code Language (Java/Javascript) • Natural Language Processing (Stemming) • Time interval for historical evaluation
3	Operations	Source Code Indexing	<p>User can select a path for source code to work.</p>
4	Operations	Source Code Indexing	<p>Source code should be parsed with structural information such as;</p> <ul style="list-style-type: none"> • FileContent • Class • Variables • Function • -ObjectKey
5	Operations	Source Code History Indexing	<p>Source code history should be indexed in structured way.</p> <p>Following fields must be created;</p> <ul style="list-style-type: none"> • CommitId • CommitMessage • ModifiedFiles
6	Operations	Adaptive Attribute Weighting	<p>An optimization algorithm must be implemented to support project specific weights on query fields.</p> <ul style="list-style-type: none"> • AAW algorithm must be implemented based on Genetic algorithms
7	Operations	Query Construction	<p>Dynamic query construction algorithm must be implemented.</p> <p>Fields and Coefficients should be selected according to index which retrieval process executed.</p>

Table 5.1 continues

8	Operations	Retrieval	<p>Queries must be executed by service calls.</p> <p>Response must include:</p> <ul style="list-style-type: none"> • Potentially buggy file name • File score
9	General	Internalization	<p>Application can support languages below;</p> <ul style="list-style-type: none"> • English • Turkish

All of these features are implemented and published.

5.1.2 BugSTAiR UI Dashboard

BugSTAiR UI dashboard module can include features about user interface and administrative reporting processes. We prepared a statement-of-work (SoW) document to identify all the needs. Feature specifications is given in Table 5.2.

Table 5.2 Features of BugSTAiR Core Services

No	Feature	Epic	Description
1	General	Internalization	<p>Application can support languages below;</p> <ul style="list-style-type: none"> • English • Turkish
2	Onboarding	Onboarding	<p>Onboarding screens should be shown to user at first login.</p> <p>Onboarding screens should have max 5 screenshots.</p>

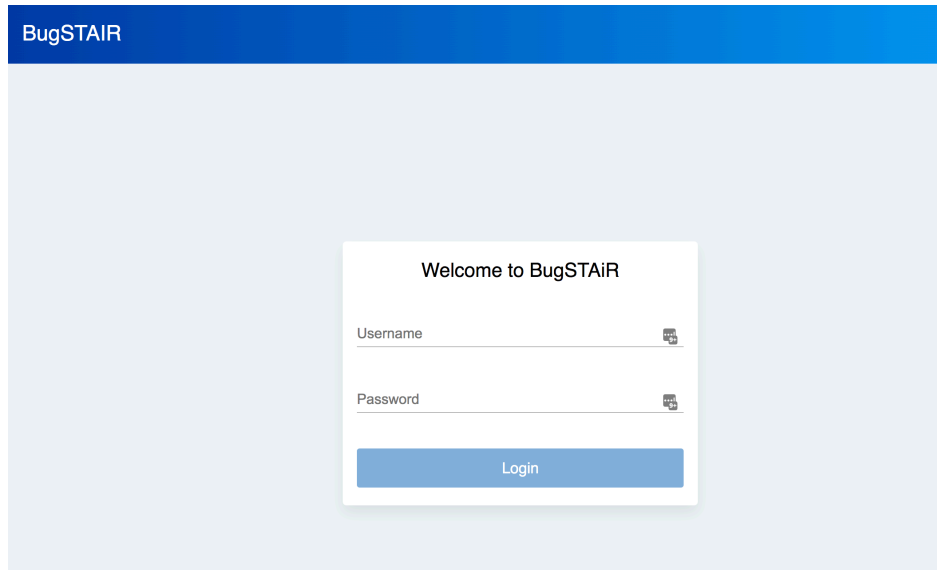
Table 5.2 continues

3	Homepage	Session Management	<p>Application should check the session if there is an active session or not.</p> <ul style="list-style-type: none"> • Navigate to login page if there is not active session. • Navigate to main page if there is an active session.
4	Homepage	Homepage - Pre login	<p>The homepage should be viewed with the following component unless a session created.</p> <ul style="list-style-type: none"> • Login • Remember Password
5	Login	Login Page	<p>Users can login with username and password. Login information must be validated on-prem database or active directory.</p>
6	Homepage	Homepage – Post Login	<p>User should be redirect to homepage if login cridentials are validated.</p> <ul style="list-style-type: none"> • User can only see projects which are already authorized. This authorization inherits from Project management system. • Project listing must be implemented in two ways such as List View and Grid View • User can switch between views by a toggle. • List must have pagination, user can navigate a page via page number
7	Homepage	Project Detail List View	<p>List view must include some basic statistics about the project on mouse hover action. These are ;</p> <ul style="list-style-type: none"> • Project summary • Bug Count • Last Index Information
8	Homepage	Project Detail Grid view	<p>Grid view must include some basic statistics about the project such as;</p> <ul style="list-style-type: none"> • Project summary • Bug Count • Search Statistics • Last Index Information

Table 5.2 continues

9	Operations	Search	<ul style="list-style-type: none"> • User can open search page by a button. • User can see previous searches in a list under the search bar. <ul style="list-style-type: none"> ◦ Previous search list must have pagination and user can navigate by page number • User can create a query by using a text area and submit query by using search button. • Search can be done in two ways such as; <ul style="list-style-type: none"> ◦ Similar bug list ◦ Buggy file prediction <p>According to the IR scores</p>
10	Operations	Search Result	<p>User can see search results according to the operation.</p> <ul style="list-style-type: none"> • If user searches for similar bugs, result list must contain previous similar bugs and file list which were fixed. • If user searches for bug prediction, result list must contain possible buggy files with score.
11	Operations	Query Feedback	<p>Developer can search its own previous queries and provide a feedback for application success.</p> <p>Developer can see predicted files according to search and select the real fixed files after the bug fixed.</p>
12	Plugin	Jira Plugin	<p>A jira plugin must be implemented which uses search and feedback services. Therefore, developers can see the search result and feedback file list via Jira field.</p>

Core services do not have any user interface related outputs. All the features implemented as independent services. Therefore, it is easy to scale up for new projects. Some screenshots are given to introduce BugSTAiR UI Dashboard application below.



The login page features a blue header with the text "BugSTAIR". Below the header is a white login form with a blue border. The form contains the following elements:

- Welcome to BugSTAIR**: A heading centered at the top of the form.
- Username**: A text input field with a small icon of a person on the right.
- Password**: A text input field with a small icon of a key on the right.
- Login**: A blue button centered at the bottom of the form.

Figure 5.1 Login Page of BugSTAiR

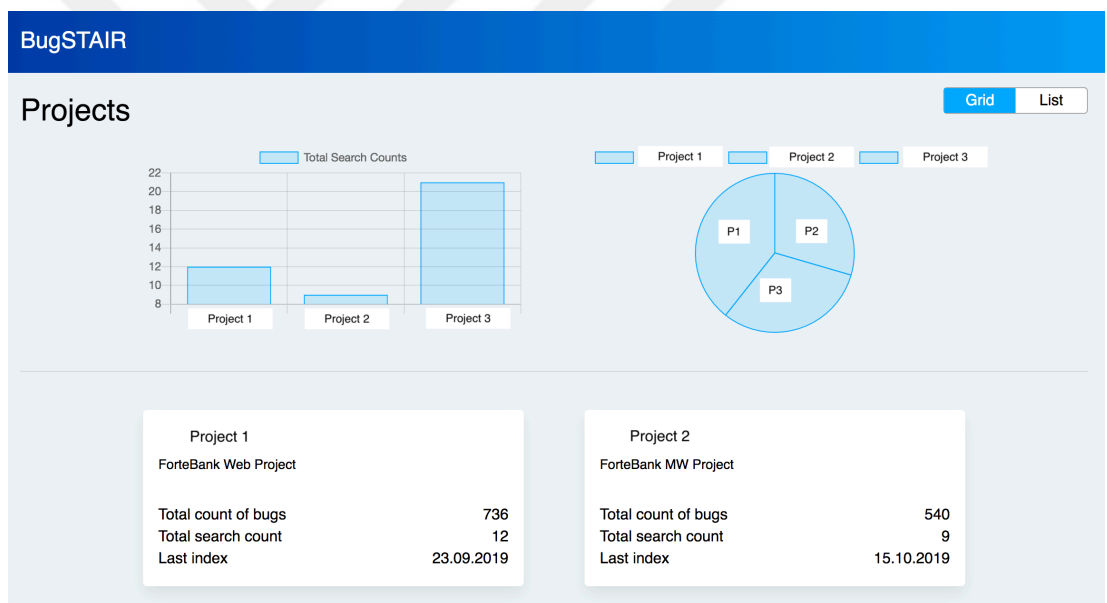


Figure 5.2 Home page of BugSTAiR

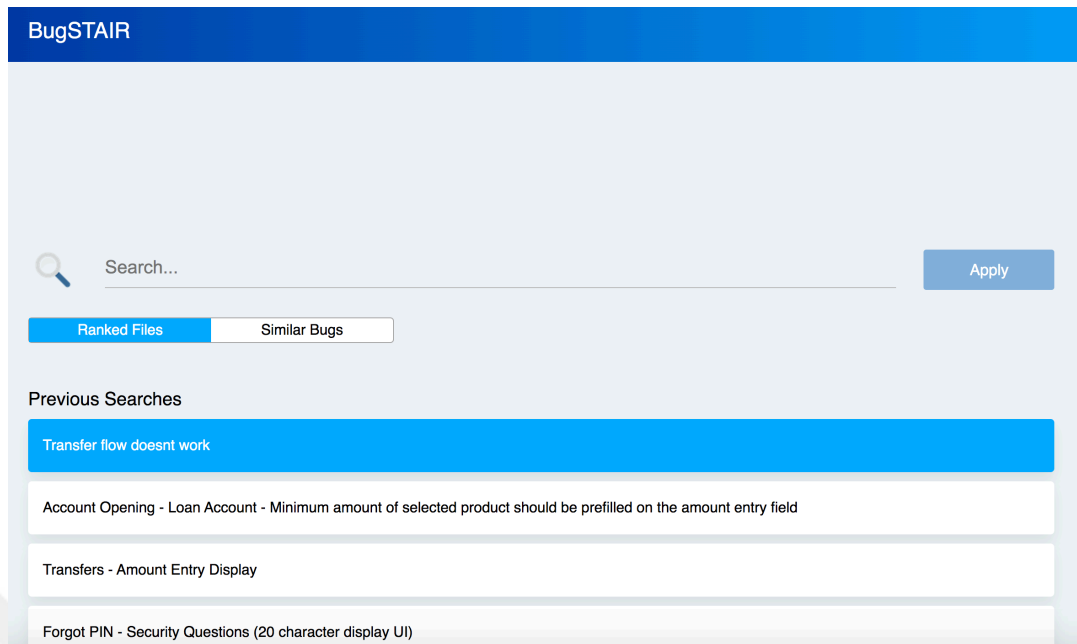


Figure 5.3 Previous search list of a user

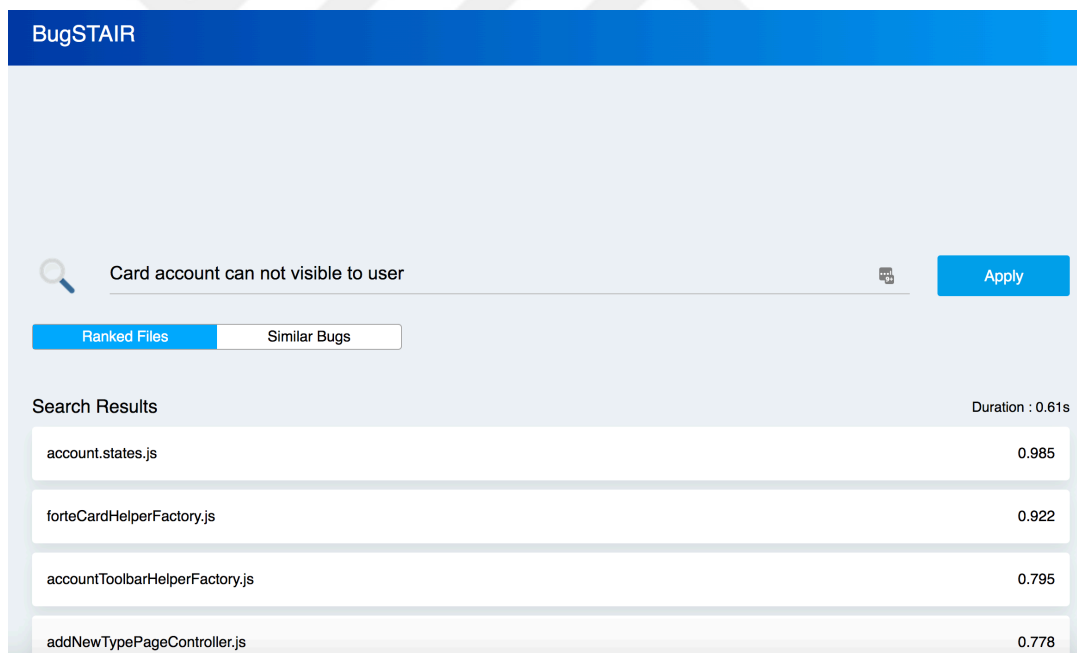


Figure 5.4 File search and search results according to file score

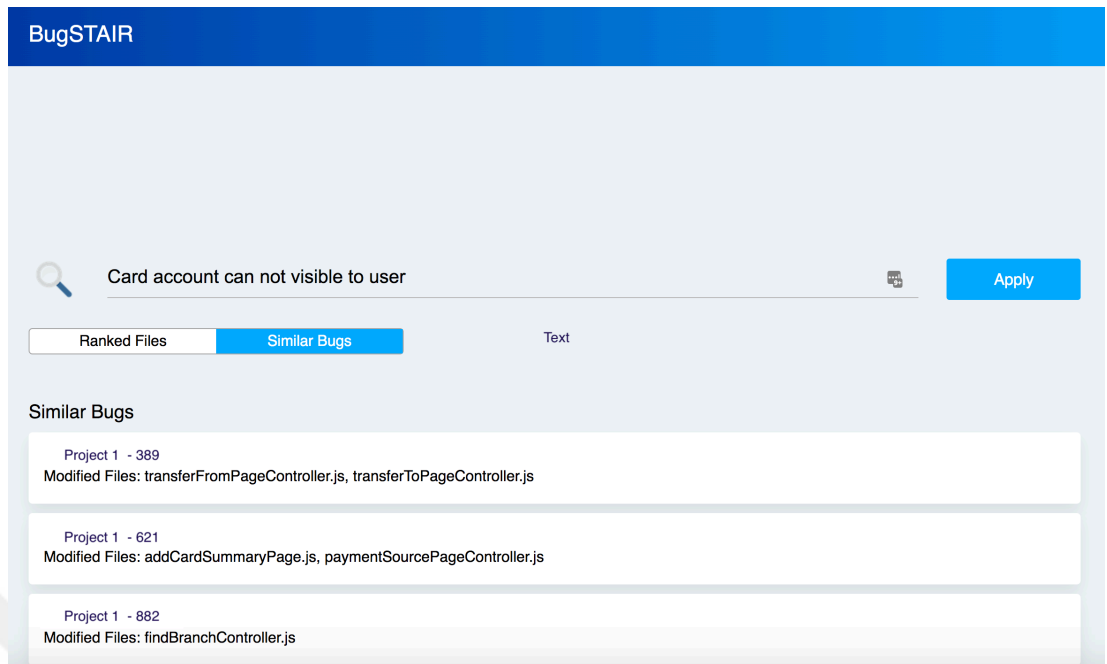


Figure 5.5 Similar bug search and result list

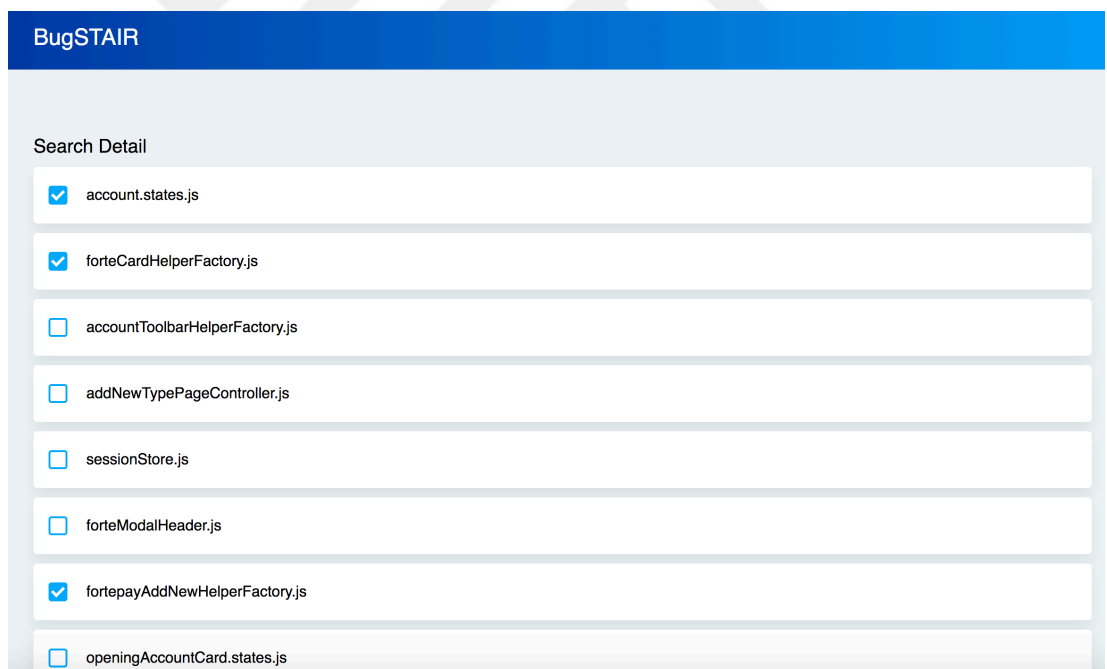


Figure 5.6 Developer feedback list for fixed issue

CHAPTER 6

EXPERIMENTAL STUDY

6.1 Subject Systems

To evaluate the success of the proposed approach, all experimental results of IR on well-known benchmark datasets such as Eclipse, AspectJ, SWT are presented. These datasets are used in the bug localization field by researchers. All of them are open-source software projects that are developed with Java. The source code and change history of the subject projects are collected from Git repository of the projects. All the bug reports which have already been fixed are collected from bug tracking systems.

Besides, a commercial web application is used to evaluate the performance of the proposed approach on JS-based application since there is no open-source benchmark dataset for JS-based applications. The detailed information about commercial web application is as follows;

- Responsive Web Application is developed with AngularJs framework.
- Development language (Method names, variables etc.) is English.
- Bug summaries and descriptions are in English.
- The number of JS file related bugs is 313.
- JIRA is used as bug tracking system.
- On-premise Git is used as source/version control system.

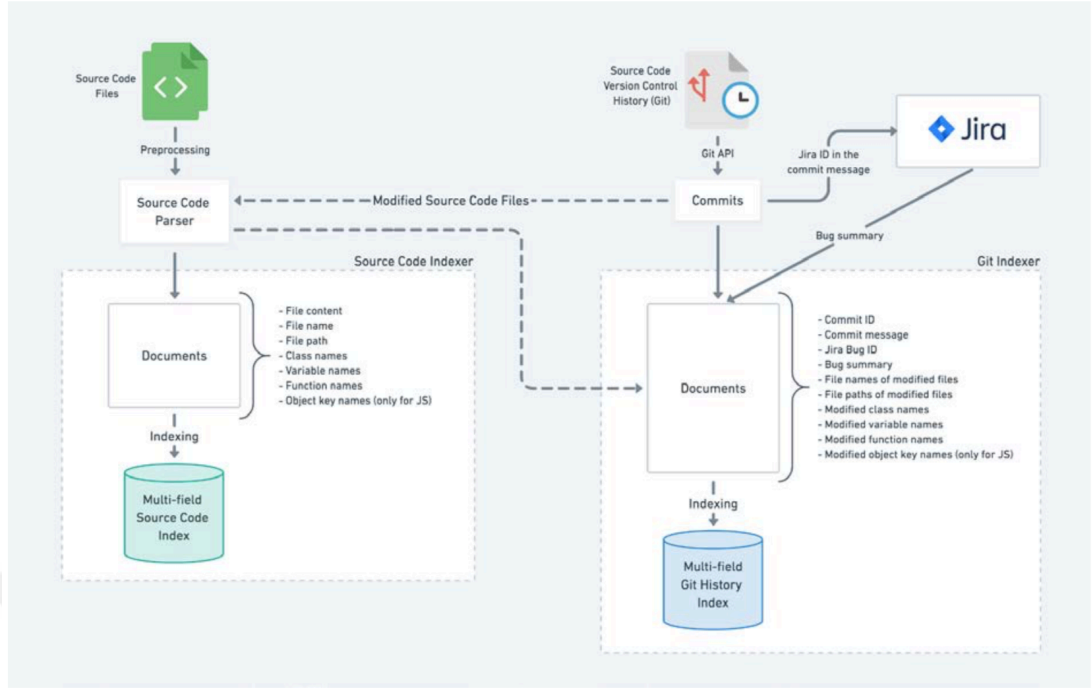


Figure 6.1 Experimental setup architecture for a software project

6.1.1 Dataset Statistics

In this section, brief information about datasets which are used to test the proposed approach is given. Some important statistics collected from datasets are shown in Table 6.1.

Table 6.1 Dataset Statistics

Dataset	Indexed Source Code Files	# of Commits	# of Bug Reports
SWT	738	33,994	98
Eclipse	12,302	37,687	1,174
AspectJ	3,692	8,291	284
Web Application	724	2,543	313

Indexed source files are different according to the dataset. Javascript files that have “js” extension are indexed in the web application dataset. Additionally, files that have “java” extension are indexed in AspectJ, SWT and Eclipse dataset. These statistics directly affect the IR result and accuracy. Also, these statistics are important for

evaluating the success of tools. Some datasets cannot be evaluated depending on the approach which the tool has adopted. Therefore, it is important to compare approaches of IRBL tools.

BLUIR+, BRTracer+, BLIA and BugLocator are well-known IRBL tools that have better accuracies on BL. Detailed comparison of these tools and BugSTAiR is shown in Table 3. Each row of the table is a property of the BL approaches. IR Method, Information Structure, Bug Similarity, Version History usage, Stack Trace usage and Adaptive Attribute Weighting are evaluated in comparison. “O” means that tools have related property and “X” means do not have related property in their approaches.

Table 6.2 Comparison of IRBL tools

Approach	BLUIR	BRTracer	BLIA	BugLocator	BugSTAiR
Published	2013	2012	2015	2012	2020
IR Method	TF.IDF	rVSM	rVSM	rVSM	Lucene
Structured Information	O	X	O	X	O
Bug Similarity	O	O	O	O	O
Version History	X	X	O	X	O
Stack Trace Analysis	X	O	O	X	X
Adaptive Attribute Weighting	X	X	X	X	O

Bug similarity is the standard feature that is used by all IRBL tools. Structured information of source code is another common feature. BRTracer+ and BLIA use information about stack traces to improve IR accuracy. Stack trace is one of the most important features because it mostly contains direct reference of buggy source. Effects of the stack trace is up to 47\% according to the reports in BLIA.

The only tool which uses source code version history except for BugSTAiR is BLIA. Source code history threatens experimental results on datasets. It is not possible to have source code histories on all benchmark datasets. Specifically, source code architecture

of Eclipse is different from any other datasets. Its source code is based on different repositories, so it is difficult to match which bug is related to which repository. To avoid this mismatch, the top three repositories that cover most bug reports are analyzed and selected. The source code history information and bug report dataset are shared in open-source platforms such as Github. This new dataset is third contribution of this study to BL research.

Current state of the IRBL tools, their approaches and details of datasets with statistics are introduced and the experimental results of the retrieval process on these datasets are explained in Section 4.4.

6.1.2 Evaluation Metrics

There are some standard evaluation metrics on IR research such as Top N Rank accuracies, Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR). All of the compared tools use the same metrics to evaluate IR results.

Top N Rank: This metric is used to calculate the number of the bug reports in which at least one source file is ranked in list of retrieval results. A higher value for this metric indicates better BL performance. [30]. Responsive Web Application is developed with AngularJs framework.

MAP: This metric is used to find average precision, is the primary metric in IR evaluation. MAP can be formulated as:

$$MAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_k) \quad (6.1)$$

MRR: This metric is based on early precision over recall logic. Reciprocal Rank is a value that is inversely proportional to the rank given by the retrieval method to a single relevant item. Shortly, the MRR is the average RR of all queries. MRR can be formulated as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (6.2)$$

These three metrics are used to evaluate the experiments of this study. Therefore, it is possible to compare our results to the results of other tools.

6.1.3 *Experimental Results*

In this section, experimental results of BugSTAiR are compared to the previous bug localization tools and the overall performance of the proposed approach are presented. The main purpose of this study is to define a generic model to localize bugs in software projects. As mentioned before, a commercial JS-based responsive web application and one of open-source JS-based projects are used as dataset to execute the proposed model. The other benchmark tools cannot work for WebApp due to development language limitations. Therefore, there are no Top N rank or MAP statistics related to WebApp for these tools. All the processes that are already defined in the proposed architecture are executed on Java-based benchmark datasets and JS-based web applications.

All the previous approaches are only work for Java-based applications such as BLIA, BRTracer+, BugLocator, BLUIR+, Locus are IRBL tools. As mentioned, there are three common metrics on BL studies such as Top 1 accuracy, MAP and MRR. BLIA is the state-of-the-art tool while we are working on proposed approach. In BL literature, BLIA had the best performance on benchmark datasets except Eclipse. Indeed, BLIA does not have any experiment on Eclipse dataset because of BLIA's

approach. BLIA uses source code history but there is not any open-sourced Eclipse dataset that includes necessary items such as source code, source code history and related bug reports. In addition, BLIA uses stack trace to have better accuracy. The effect of the stack trace on experimental results is over 40% according to BLIA's statistics. Therefore, success of the tool over other tools related to this input, but this input may not be ready for many benchmark or industrial datasets. The overall experimental results of all these tools and BugSTAiR are summarized in Table 3.

Table 6.3 Comparison of experimental results

Dataset	Approach	Top 1%	Top 5%	Top 10%	MAP	MRR
SWT	BLIA	67.3	86.7	89.8	0.65	0.75
	BLUIR+	56.1	76.5	87.8	0.58	0.66
	BRTracer+	46.9	79.6	88.8	0.53	0.60
	BugLocator	39.8	67.4	81.6	0.45	0.53
	BugSTAiR	70.41	80.61	83.67	0.69	0.74
	DNNLoc	35.2	69.0	80.3	0.45	0.37
	Locus	64.3	84.7	91.8	0.64	0.73
Eclipse	BLIA	N/A	N/A	N/A	N/A	N/A
	BLUIR+	32.9	56.2	65.4	0.33	0.44
	BRTracer+	32.6	55.9	65.2	0.33	0.43
	BugLocator	29.14	53.76	62.60	0.22	0.41
	BugSTAiR	39.52	53.06	58.09	0.43	0.46
	DNNLoc	45.8	70.5	78.2	0.51	0.41
	Locus	N/A	N/A	N/A	N/A	N/A
AspectJ	BLIA	41.5	71.1	80.6	0.39	0.55
	BLUIR+	33.9	52.4	61.5	0.25	0.43
	BRTracer+	39.5	60.5	68.9	0.29	0.49
	BugLocator	30.8	51.1	59.4	0.22	0.41
	BugSTAiR	42.3	63.4	70.2	0.43	0.51
	DNNLoc	47.8	71.2	85.0	0.52	0.32
	Locus	25.0	56.6	63.9	0.32	0.38
Tomcat	BLIA	N/A	N/A	N/A	N/A	N/A
	BLUIR+	N/A	N/A	N/A	N/A	N/A
	BRTracer+	N/A	N/A	N/A	N/A	N/A
	BugLocator	N/A	N/A	N/A	N/A	N/A
	BugSTAiR	55.11	80.4	82.77	0.61	0.65
	DNNLoC	53.9	72.9	80.4	0.60	0.52
	Locus	53.9	77.7	81.9	0.57	0.64
angular-translate	BugSTAiR	50.81	80.33	85.25	0.46	0.50
Web Application	BugSTAiR	40.23	62.34	72.63	0.46	0.50

Columns in Table 6.3 are Dataset, Approach, Top 1 %, Top 5 %, Top 10%, MAP and MRR. Dataset column indicates names of benchmark datasets that the proposed approach has already experimented. Approach column indicates names of tools that proposed approach is compared. The Top 1% value is the ratio of the number of queries in which the first ranked source code file is the correct file to the total number of queries. The Top 5% value is the ratio of the number of queries in which any of the first five ranked source code files in are correct files to the total number of queries. The Top 10% value is the ratio of the number of queries in which any of the first ten ranked source code files in are correct files to the total number of queries. MAP and MRR columns are common IRBL metrics that are mentioned in Chapter 6.1.2.

The experimental result shows that BLIA's Top 1% is 67.3% in SWT dataset, and this score makes difference among BugLocator, BLUIR and BRTracer. MAP score is 0.65 and MRR score of BLIA is 0.75 BugSTAiR locates more bugs in the first ranked source file than BLIA in this dataset. Therefore, BugSTAiR's Top 1% score is better than BLIA. MAP score is also better than BLIA. MRR scores of these tools are very close to each other. Performance of BL tools are different in Eclipse dataset. BLUIR+ has 32.9% in Top 1% score, and this score was the best in IRBL tools before BugSTAiR. BLIA and Locus do not have any experiments on Eclipse dataset. BugSTAiR's Top 1% score is 39.52%, MAP is 0.43 and MRR is 0.46 on this dataset. Therefore, BugSTAiR outperforms all IRBL tools on Eclipse. The general view of tool performances is the same on AspectJ dataset. BLIA was the best performance in IRBL tools by 41.5% Top 1% score, 0.39 MAP and 0.55 MRR score. BugSTAiR locates more bugs in the first ranked result than BLIA. Experimental results of BugSTAiR in AspectJ in Top 1%, MAP and MRR are 42.3%, 0.43 and 0.51 respectively. Tomcat dataset has been used in recent studies. Therefore, older studies do not have an experimental result on this data set. Locus had the best performance scores in IRBL tool by 53.9% on Top1%, 0.57 on MAP and 0.64 MRR scores. BugSTAiR's performance metrics are 55.1%, 0.61 and 0.65 on Top1%, MAP and MRR respectively. Hence, BugSTAiR has the best performance in Tomcat dataset. Since the last two data sets are developed with the JS-based software development language, they do not have any experimental results with other bug localization tools.

In short, experimental results show that BugSTAiR performs better than all the previous studies that use IRBL. BugSTAiR Top 1 rank is 2% and MAP is 10% better than BLIA metrics on AspectJ. BugSTAiR has localized 4.6% bugs in Top 1 and its precision is 6.1% better than BLIA in SWT. As mentioned, there is no performance metric on Eclipse for BLIA, so BLUIR+ had the best scores. Then, BugSTAiR's performance is compared to BLUIR+. Experimental results show that performance of BugSTAiR is 20% better than BLUIR+ in Top 1 metric and its precision is also 30% better than BLUIR+.

On the other hand, there is another tool on the comparison table which did not compared to IRBL tools, DNNLoc. DNNLoc is a BL tool that uses a deep neural network while locating a buggy source. Experimental results of DNNLoc are better than BugSTAiR in both Eclipse and AspectJ on Top 1 and MAP metrics. These metrics are essential, but MRR scores of DNNLoc are remarkably less than all other tools. Ideally, the MRR score should be better than the MAP score in the IR process. In detail, the MAP metric deals with the first predicted item and true positive values over all positives. MRR metric deals with the only actual rank of the prediction. MRR metric could be less than MAP if many predicted results are not found in the actual results. Unless a tool is not consistent for all queries in given dataset, the tool can cause extra costs in BL processes. According to this fact, BugSTAiR could be more preferred tool than DNNLoc.

There are some statistics about the proposed approach that cannot compared to the other tools. Execution time of each processes which are source code indexing, change history indexing and query retrieval processes is calculated. Number of source files and number of changes affect the execution time. All of these statistics are shown in Table 6.4.

Table 6.4 Execution Time Statistics of BugSTAiR

Dataset	# of Source Files	# of Commits	Indexing Time (sec)	Avg IR Time (sec)
SWT	738	33,994	618.8	2.01
Eclipse	12,302	37,687	11,544	0.71
AspectJ	3,692	8,291	287	1.7
Web Application	724	2,543	35.6	0.38
Tomcat	2,485	62,783	24,348	1.21
angular-translate	48	1,712	219	0.67

6.1.4 Threats to Validity

This section considers threats to validity. Four types of them are explained: Threats to internal validity, threats to external validity, threats to construct validity and reliability.

Threats to internal validity is biases that may be done by experimenters. In the proposed approach, the same dataset as BugLocator and BLIA have used. These are well-known datasets which are used to minimize threats to internal validity. The source files and change histories are downloaded from Git repositories of the projects. Afterwards, all extended properties such as fixed files and commit logs are verified for each dataset.

Threats to external validity is about the generalizability of the results. Most bug localization tools only work for well-known open-source datasets. Our approach is tested on four datasets of different sizes, different domains and different languages. One of these datasets is from an internally developed project for commercial use. Therefore, our approach is generalizable to any other open-source or commercial project with different languages. A potential threat to validity is the quality of bug reports. Bug reports contain many crucial information about the issue for developers to fix the bugs. If a bug report has misleading information or does not provide enough information, the accuracy of the BugSTAiR is affected poorly.

Threats to construct validity is about the qualification of the evaluation metrics. In our experiments, three evaluation metrics such as Top N rank, MAP and MRR are used. These metrics have been widely used for bug localization benchmarks and are well-known IR metrics. Therefore, it is obvious that our research has strong construct validity.

In previous bug localization tools, various combinations of control parameters have been used to find the best accuracy for each project. Every parameter has been defined according to the number of experiments for each dataset. In our approach, the AAW process is proposed to optimize the control parameters. All the parameters are automatically selected via GA. Therefore, there are no heuristic or experimental tests in our approach.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Software quality and maintainability are getting more important in the last decade due to improvements on software development technologies. Quality of a software is directly related with processes and practices that are used in SDLC. Typical SDLC has iterative phases such as requirement analysis, design, development, testing and maintenance. Software engineering discipline defines several activities and techniques for each phases of these lifecycle. Although following all defined activities, all software's have bugs in production. Therefore, dealing with bugs is a process that always in SDLC.

Bugs can be reported by three different profiles, such as developers, testers, and end-users. Then, a developer must locate the buggy source to fix it. This process is generally difficult and time-consuming. Many researchers studied in this area to automate the bug localization process. Bug localization is one of the critical points in bug fixing processes. Several approaches are proposed to improve bug localization efficiency and accuracy in the last decade. Information retrieval-based BL, machine learning based BL, and neural network-based BL are common approaches in BL literature. All of the previous studies include state-of-the-art algorithms works for only Java-based applications. There is no proposed solution to other popular software development languages such as Javascript, Kotlin, and Swift. Besides, all of these studies are academic purposes. Therefore, there is not any study or tool that is ready for industrial use.

In this research, the proposed approach uses information retrieval and genetic algorithms on both JS-based web applications and Java-based back-end applications. It is the first BL work that works for JS-based web applications by using IR and ML to the best of our knowledge. Lack of the bug report dataset of open-source web

applications has made us use one of our commercial web applications. Its bug report dataset is used to experiment the proposed approach. Thus, there is no result to compare the success of the proposed approach in this area. On the other hand, results of experiments show that BugSTAiR has promising performance on Java-based applications, so BugSTAiR outperforms any other BL tool. The experimental results indicate that BugSTAiR has better performance than BLUIR+ and BLIA. The proposed tool localized 20% bugs in Eclipse, 4.6% bugs in SWT and 2% bugs in AspectJ. Besides our system has better performance (on the MAP metric) than any other tool with all datasets. MAP metrics of BugSTAiR are 6.1%, 10% and 30% higher compared to BLIA and BLUIR+.

In addition to this, BugSTAiR is the first generic BL tool with the AAW process, which is the most valuable contribution to the state-of-the-art on BL. Adaptation step prevents numerous manual experiments to reach optimum weights for all datasets. This generic implementation of the BL process provides us to enlarge our datasets easily.

At last, BugSTAiR is the first BL tool that is ready for industrial usage. All of the previous tools are based on academic purposes.

7.2 Future Work

In the future, we would like to integrate image processing features to handle screenshots that are taken when a bug or error is occurred. As it is possible to extract more valuable text information from images to localize bugs, this feature will help us to improve localization accuracy in two ways. First, we can localize only JS files in web applications. Second, it will be possible to localize UI related bugs more accurately with the help of this feature. In addition to this, we would like to integrate the proposed model with ML algorithms such as clustering. It is possible for a bug to be related with another bug which was fixed before. Therefore, clusters which are created according to the relevance of textual similarity between bug reports can help to improve the accuracy of IR results. Finally, configurable software systems and microservice architectures will be preferred in both front-end and back-end in the

future. If there are any open-source projects like this, we will test the BugSTAiR and share experimental results.



REFERENCES

- Apache Lucene. (2020). Retrieved March 4, 2020, from <http://lucene.apache.org/core/>.
- Developer Survey Results. (2020). Retrieved March 4, 2020, from <https://insights.stackoverflow.com/survey/2019/>
- Du, H., Wang, Z., & Zhan W. (2018). Elitism and distance strategy for selection of evolutionary algorithms. *IEEE Access Digital Object Identifier*, 44531-44541
- Ersahin, B., Aktas, O., Kilinc D., & Ersahin, M (2019). A hybrid sentiment analysis method for Turkish, *Turkish Journal of Electrical Engineering & Computer Science*, 1780 – 1793
- Goldberg, D.E. (1989). *Genetic algorithms in search optimization and machine learning*. Boston: Addison-Wesley Longman Publishing Company.
- Gopinath, D., Zaeem, R.N., & S. Khurshid. (2012). Improving the effectiveness of spectra-based fault localization using specification, *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference*, 40-49
- Hovemeyer, D., & Pugh, W. (2004). Finding bug is easy, *ACM Sigplan Notices*, 39, 92-106
- Kim, D., Tao, Y., & Kim S. (2013). Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 1597-1610
- Kilinc, D. (2019). A spark-based big data analysis framework for real-time sentiment prediction on streaming data. *Journal of Software: Practice and Experience*, 49, 1352-1364
- Kilinc, D., Yucalar, F., Borandag, E., & Aslan, E. (2016). Multi-level re-ranking approach for bug localization. *Expert Systems*, 33, (3), 286-294
- Moreno, L., Treadway, J., & Marcus, A. (2014). On the use of stack traces to improve text retrieval-based bug localization. *IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, 151-160

- Nguyen, A.T., Nguyen, T.T., & Al-Kofahi, J. (2011). A topic-based approach for narrowing the search space of buggy files from a bug report, *26th IEEE/ACM International Conference on Automated Software Engineering*, 263-272
- Polisetty, S., Miranskyy, A., & Başar A. (2017). On usefulness of the deep-learning-based bug localization models to practitioners, *15th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*, 16-25
- Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., & Rajlich, V. (2006). Combining probabilistic ranking and latent semantic indexing for feature identification, *14th IEEE International Conference on Program Comprehension (ICPC 2006)*, 137-146
- Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., & Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, *IEEE Transactions on Software Engineering*, 33, 420-342.
- Rao, S., & Kak, A. (2011). Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. *15th IEEE International Conference on Program Comprehension, ICPC'07*, 37-48.
- Raulji, J.K., & Jatinderkumar, R.S. (2016). Stop-word removal algorithm and its implementation for Sanskrit language. *International Journal of Computer Applications*, 15-17
- Ricardo, B.Y., & Berthier, R.N. (2013). *Modern information retrieval*. New York: ACM Press.
- Saha, R.K., Lease, M., Khurshid, S., & Perry, D.E. (2013) Improving bug localization using structured information retrieval. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference*, 345–355.
- Schroter, A., Bettenburg, N., & Premraj, R. (2010). Do stack traces help developers fix bugs?, *7th IEEE Working Conference*, 118-121
- Thisted, R.A. (1988). *Elements of statistical computing: Numerical computation*. Boston: Addison-Wesley Longman Publishing Company.

- Voorhees, E.M., & Harman, D.K. (2002). Chapter appendix: common evaluation measures. *The Eleventh Text Retrieval Conference*. Maryland, USA: National Institute for Standards and Technology.
- Wang, S., Lo D., & Lawall J. (2014). Compositional vector space models for improved bug localization, *International Conference on Software Maintenance and Evolution (ICSME'14)*, 171-180
- Xiao, Y., Keung, J., Mi, Q., & Bennin K.E. (2017). Improving bug localization with an enhanced convolutional neural network, *24th Asia-Pacific Software Engineering Conference (APSEC)*, 338-347
- Youm, K. C., Ahn, J., & Lee, E. (2017). Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82, 177-192.
- Zhou, J., Zhang, H., & Lo, D. (2012). Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. *Software Engineering (ICSE), 2012 34th International Conference on, IEEE*, 14–24 .

APPENDICES

APPENDIX: LIST OF ACRONYMS

Acronym	Definition
VSM	Vector Space Model
LDA	Latent Dirichlet Allocation
AAW	Adaptive Attribute Weighting
IR	Information Retrieval
IRBL	Information Retrieval-Based Bug Localization
MAP	Mean Average Precision
MRR	Mean Reciprocal Rank
GA	Genetic Algorithm
LSA	Latent Semantic Analysis