A dissertation submitted to the
Fakultät für Informatik und Automatisierung
Technische Universität Ilmenau

Utilizing Traceable Software Artifacts to Improve Bug Localization

for the degree of

DOKTOR-INGENIEUR (DR.-ING.)

by

Dipl.-Inf.
Michael Rath

born October 4, 1982
in Neuhaus am Rennweg, Germany

accepted on the recommendations of

Prof. Dr.-Ing. Patrick Mäder, TU Ilmenau
Prof. Dr.-Ing. habil. Armin Zimmermann, TU Ilmenau
Univ.-Prof. Dr.-Ing. habil. Matthias Riebisch, Universität Hamburg

DOI : 10.22032/dbt.53717
URN : urn:nbn:de:gbv:ilm1-2022000360

Day of submission: November 25, 2021
Day of scientific debate: October 24, 2022

Abstract

The development of software systems is a very complex task. Quality assurance tries to prevent defects – software bugs – in deployed systems, but it is impossible to avoid bugs all together, especially during development. Once a bug is observed, typically a bug report is written. It guides the responsible developer to locate the bug in the project's source code, and once found to fix it. The bug reports, along with other development artifacts such as requirements and the source code are stored in software repositories. The repositories allow to create relationships – trace links – among contained artifacts. Establishing this traceability is demanded in many domains, such as safety related ones like the automotive and aviation industry, or in development of medical devices.

However, in large software systems with thousands of artifacts, especially source code files, manually locating a bug is time consuming, error-prone, and requires extensive knowledge of the project. Thus, automating the bug localization process is actively researched since many years. Further, manually creating and maintaining trace links is often considered as a burden, and there is the need to automate this task.

Multiple studies have shown, that traceability is beneficial for many software development tasks. This thesis presents a novel bug localization algorithm utilizing traceability. The project's artifacts and trace links are used to create a traceability graph. Afterwards, this graph is analyzed to locate defective source code files for a given bug report. Since the existing set of trace links of a project is possibly incomplete, another algorithm is prosed to augment missing links. This *augmentation* algorithm is fully automated, project independent, and derived from a project's development workflow.

An evaluation on more than 32,000 bug reports from 27 open-source projects shows, that incorporating traceability information into bug localization significantly improves the bug localization performance compared to two state of the art algorithms. Further, the trace link augmentation approach reliably constructs missing links and therefore simplifies the required trace maintenance.

Zusammenfassung

Die Entwicklung von Softwaresystemen ist eine komplexe Aufgabe. In der Qualitätssicherung wird versucht auftretendene Softwarefehler – bugs – in ausgelieferten Systemen zu vermeiden. Dennoch können Fehler nie ausgeschlossen werden, besonders während der Entwicklung. Sobald ein Softwarefehler entdeckt wird, wird typischerweise ein Fehlerbericht (*bug report*) erstellt. Dieser dient als Ausgangspunkt für den verantwortlichen Entwickler den Fehler im Quellcode des Programms zu finden und letztendlich diesen zu beheben (*bug fixing*). Alle Fehlerberichte sowie weitere Softwareartefakte, z.B. Anforderungen und der Quellcode selbst, werden zusammen in Software Repositories abgelegt. Diese erlauben außerdem die Artefakte miteinander via *trace links* zur Nachvollziehbarkeit zu verknüpfen. Die Erstellung dieser Nachvollziehbarkeit zwischen den Artfakten ist in vielen Anwendungsdomänen sogar vorgeschrieben. Dazu zählen u.a. die Luftfahrt- und Automobilinstustrie, sowie die Entwicklung von medizinischen Geräten.

Jedoch ist das Auffinden von Softwarefehlern in großen Systemen mit tausenden Artefakten, insbesondere Quellcodedateien, eine anspruchsvolle, zeitintensive und fehleranfällige Aufgabe, welche eine umfangreiche Projektkenntnis erfordert. Aus diesem Grund wird seit mehreren Jahren aktiv an der Automatisierung dieses Prozesses geforscht. Weiterhin wird die manuelle Erstellung und Pflege von trace links als Belastung empfunden und sollte somit ebenfalls weitgehend automatisiert werden.

Eine Vielzahl von Studien hat gezeigt, dass eine etablierte Nachvollziehbarkeit vorteilhaft für verschiedenste Softwareentwicklungsaufgaben ist. In dieser Arbeit wird ein neuartiger Algorithmus zum Auffinden von Softwarefehlern vorgestellt, der aktiv die erstellten Verknüpfung zwischen Softwareartefakten ausnutzt. Hierzu werden die Artefakte und deren Beziehungen herangezogen um einen Nachvollziehbarkeitsgraphen zu erstellen. Anschließend wird der Graph analysiert um fehlerhafte Quellcodedateien anhand eines gegebenen Fehlerberichtes zu finden. Jedoch muss angenommen werden, dass nicht alle notwendigen Verknüpfungen zwischen den Softwareartefakten eines Projektes vorhanden sind. Aus diesem Grund wird ein vollautomatisierter, projektunabhängiger Ansatz vorgestellt, der diese fehlenden trace links erstellt. Die Grundlage zur Entwicklung dieses Algorithmus ist der typische Arbeitsablauf eines Projektes.

Der vorgeschlagene Ansatz wurde mit mehr als 32.000 Fehlerberichten von 27 Open-Source Projekten evaluiert. Die erzielten Ergebnisse zeigen, dass die Einbeziehung von trace links zwischen Softwareartefakten signifikant das Auffinden von Fehlern im Quellcode im Vergleich zu zwei Algorithmen (Stand der Technik) verbessert. Weiterhin konnte gezeigt werden, dass notwendige jedoch fehlenden trace links zwischen Softwareartefakten zuverlässig erstellt werden können und somit der Aufwand zum Pflegen der Nachvollziehbarkeit vereinfacht wird.

Contents

Abstract i

Zusammenfassung iii

Contents v

List of Publications 1
- Publications Included in this Thesis . . . 1
- Related Publications . . . 2

Contribution Statement 5

1. Introduction 7

2. Background 11
- 2.1. Defining Essential Terms . . . 11
- 2.2. Developing Software in an Agile Way . . . 12
- 2.3. Software Traceability . . . 13
- 2.4. Common Tools Utilized in the Agile Development Process . . . 14
  - 2.4.1. Managing Requirements and Bug Reports in Issue Tracking Systems . . . 15
  - 2.4.2. Managing Source Code Files in Version Control Systems . . . 20
  - 2.4.3. Creating Trace Links between Artifacts in Jira and Git . . . 22
- 2.5. Localizing Bugs using Information Retrieval Techniques . . . 23
  - 2.5.1. Indexing the Source Code File Corpus . . . 26
  - 2.5.2. Constructing a Query from a Bug Report . . . 26
  - 2.5.3. Retrieve and Rank Source Code Files . . . 27

3. State of the Art 29
- 3.1. Bug Localization . . . 29
  - 3.1.1. Datasets and Collections of Projects used to Evaluate Approaches 32
  - 3.1.2. Comparison of IR-based Bug Localization Algorithms . . . 33
  - 3.1.3. Internal Structure of Bug Localization Algorithms . . . 37
- 3.2. Localizing Features and Recovering Trace Links . . . 38
- 3.3. Criticizing the State of the Art . . . 40

4. A Holistic Approach to Improve IR-based Bug Localization 41
4.1. Outlining the Idea 41
4.1.1. Creating a Dataset to Evaluate Bug Localization Algorithms . 43
4.1.2. Designing a Bug Localization Algorithm utilizing Traceability 43
4.1.3. Augmenting the Issue-to-Commit Trace Link Set 43
4.2. Constructing an Artifact Model 44
4.2.1. Describing Contained Artifacts 44
4.2.2. Describing Artifact Relations 44
4.3. Summary 45

5. Mining Software Repositories to Create Holistic Datasets 47
5.1. Selecting Projects for Mining 48
5.2. Collecting Project Artifacts from Multiple Repositories 49
5.2.1. Analyzing a Project's Issue Tracking System 49
5.2.2. Analyzing a Project's Version Control System 51
5.3. Organizing Project Artifacts in Unified Storage 52
5.4. Key Figures of the Created Dataset 53
5.5. Defining the Mined Dataset 56
5.6. Summary 56

6. The ABLoTS Bug Localization Approach 57
6.1. Motivating Example to Leverage ITS Project Data for Bug Localization 58
6.2. Multi-Component IR-Based Bug Localization Algorithms 59
6.2.1. Dissecting Existing Algorithms 59
6.2.2. Analyzing Algorithms used in Similar Bug Report Components 60
6.3. Designing a Similar Issue Component - TraceScore 61
6.3.1. Selecting Project Artifacts 62
6.3.2. Textual Processing 63
6.3.3. Constructing a Traceability Graph 63
6.3.4. Analyzing the Traceability Graph 65
6.4. Refining the Source Code Structure Component - LuceneScore 67
6.5. Utilizing TraceScore in a Bug Localization Algorithm - ABLoTS . . . 68
6.5.1. Internal Structure of ABLoTS 68
6.5.2. The Composer Component of ABLoTS 68
6.6. Summary 72

7. Automatically Augmenting Incomplete Issue-to-Commit Trace Links 73
7.1. Analyzing Existing Issue-to-Commit Trace Links 74
7.2. Motivating Example to Automatically Tag Commit Messages 77
7.3. Developing a Commit Message Tagging Model 77
7.3.1. Analyzing the Development Process 79
7.3.2. Analyzing the Projects' Stakeholder Activities 82

7.4. Creating a Trace Link Classifier 83
7.4.1. Deriving Process Related Features 83
7.4.2. Deriving Textual Similarity Features 84
7.4.3. Creating Feature Vectors 85
7.5. Summary 86

8. Evaluation **89**
8.1. Research Questions 89
8.2. Introducing the Evaluation Datasets 90
8.2.1. Creating a Gold Standard Dataset GS 90
8.2.2. Creating Datasets with Reduced Trace Link Sets RED_i 96
8.2.3. Augmenting Trace Link Sets to Create Datasets AUG_i 97
8.3. Evaluation Metrics 98
8.3.1. Top@k 98
8.3.2. Mean Average Precision (MAP) 99
8.3.3. Mean Reciprocal Rank 99
8.3.4. Cliff's delta 99
8.3.5. Precision, Recall, and F-score 100
8.4. Experiment I: Effectiveness of Similar Issue Component TraceScore (RQ-1) 102
8.5. Experiment II: Impact of TraceScore Parameterization (RQ-2) 103
8.6. Experiment III: Effectiveness of IR-Based Bug Localization Algorithm using TraceScore 105
8.6.1. Training and Running ABLoTS on a Project 105
8.6.2. Results 110
8.7. Experiment IV: Effectiveness of Trace Link Set Augmentation (RQ-4) 111
8.7.1. Training and Running the Trace Link Set Augmentation Classifier 111
8.7.2. Results 114
8.8. Experiment V: Effectiveness of IR-Based Bug Localization Algorithms on Projects with Augmented Trace Link Sets (RQ-5) 115

9. Discussion **119**
9.1. RQ-1 - Effectiveness of Similar Issue Component TraceScore 119
9.2. RQ-2 - Impact of TraceScore Parameterization 120
9.3. RQ-3 - Effectiveness of an IR-based Bug Localization Algorithm using TraceScore 121
9.4. RQ-4 - Effectiveness of Trace Link Set Augmentation 125
9.4.1. Evaluating Additional Trace Link Removal Settings 126
9.5. RQ-5 - Effectiveness of IR-based Bug Localization Algorithms on Projects with Augmented Trace Link Sets 129

9.6. RQ-6 - Limitations of Studied Approaches 134
9.6.1. ABLoTS Requires a Project History 134
9.6.2. Augmenting Issue-to-Commit Trace Link Set in Large Projects 134
9.6.3. Issue-to-Commit Trace Link Set Augmentation Requires Existing Links 135
9.7. Threats to Validity 135
9.7.1. Project Selection and Dataset Creation 136
9.7.2. Experiments I-V 137
9.7.3. Specifically for Experiments I and II 138
9.7.4. Specifically for Experiment III 138
9.7.5. Specifically for Experiment IV 138

10. Conclusion and Future Work 139
10.1. Summary 139
10.2. Future Work 141

A. Appendix I
A.1. Evaluating Different Temporal Settings for TraceScore I
A.2. Evaluation Details for Similar Issue Components IV
A.3. Evaluation Details for Bug Localization Algorithms VI

List of Tables XI

List of Figures XIII

List of Abbreviations XV

References XVI

List of Publications

This introduction chapter of the thesis lists the included and related publications.

Publications Included in this Thesis

I. **The IlmSeven Dataset** (Rath, Rempel, and Mäder 2017)
Michael Rath, Patrick Rempel, and Patrick Mäder
Requirements Engineering (RE), pages 516–519, 2017

II. **Analyzing requirements and traceability information to improve bug localization** (Rath, Lo, and Mäder 2018)
Michael Rath, David Lo, and Patrick Mäder
Mining Software Repositories (MSR), pages 442–453, 2018

III. **Traceability in the wild: automatically augmenting incomplete trace links** (Rath et al. 2018)
Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder
International Conference on Software Engineering (ICSE), pages 834-845, 2018

IV. **The SEOSS 33 dataset — Requirements, bug reports, code history, and trace links for entire projects** (Rath and Mäder 2019b)
Michael Rath and Patrick Mäder
Data in brief, Vol. 25, p. 104005, 2019.

V. **SpojitR: Intelligently link development artifacts** (Rath, Tomova, and Mäder 2020)
Michael Rath, Mihaela T. Tomova, and Patrick Mäder
International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020

Related Publications

VI. **Are Graph Query Languages Applicable for Requirements Traceability Analysis?** (Rath et al. 2017)
Michael Rath, David Akehurst, Christoph Borowski, and Patrick Mäder
Working Conference on Requirements Engineering: Foundations for Software Quality (REFSQ), vol. 1796, 2017

VII. **State of the art of traceability in open-source projects** (Rath, Goman, and Mäder 2017)
Michael Rath, Maxim Goman, and Patrick Mäder
Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen, pages 8-11, 2017

VIII. **Preprocessing texts in issue tracking systems to improve IR techniques for trace creation** (Tomova, Rath, and Mäder 2017)
Mihaela T. Tomova, Michael Rath, and Patrick Mäder
Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen, pages 17-20, 2017

IX. **Lessons learned from analyzing requirements traceability using a graph database** (Goman, Rath, and Mäder 2017)
Maxim Goman, Michael Rath, and Patrick Mäder
Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen, pages 27-30, 2017

X. **Use of trace link types in issue tracking systems** (Tomova, Rath, and Mäder 2018)
Mihaela T. Tomova, Michael Rath, and Patrick Mäder
International Conference on Software Engineering (ICSE) Companion volume, pages 181-182, 2018

XI. **Influence of Structured Information in Bug Report Descriptions on IR-Based Bug Localization** (Rath and Mäder 2018)
Michael Rath and Patrick Mäder
Software Engineering and Advanced Applications (SEAA), pages 26-32, 2018

XII. **Selecting open source projects for traceability case studies** (Rath, Tomova, and Mäder 2019)
Michael Rath, Mihaela T. Tomova, and Patrick Mäder
Working Conference on Requirements Engineering: Foundations for Software Quality (REFSQ), pages 229-242, 2019

XIII. **Structured information in bug report descriptions - influence on IR-based bug localization and developers** (Rath and Mäder 2019a)
Michael Rath and Patrick Mäder
Software Quality Journal, vol. 27, no. 3, pages 1315–1337, 2019

XIV. **Request for comments: Conversation patterns in issue tracking systems of open-source projects** (Rath and Mäder 2020)
Michael Rath and Patrick Mäder
Symposium On Applied Computing (SAC), 2020

XV. **Did you remember to test your tokens?** (Gonzalez, Rath, and Mirakhorli 2020)
Danielle Gonzalez, Michael Rath, and Mehdi Mirakhorli
Mining Software Repositories (MSR), pages 232-242, 2020

XVI. **Cutting through the jungle: Disambiguating model-based traceability terminology** (Holtmann et al. 2020)
Jörg Holtmann, Jan-Philipp Steghöfer, Michael Rath, and David Schmelter
Requirements Engineering (RE), pages 8-19, 2020

Contribution Statement

Collaborating with other researchers is vital and a central aspect in academics. Therefore all papers in this thesis have been co-authored with other researchers. The authors' individual contributions to papers I-V are as follows.

Paper I
For the first paper, Patrick Mäder proposed the idea of creating a new dataset suitable for traceability research. Initial code fragments to automatically mine projects were provided by Patrick Rempel, which were previously used in his own research. Eventually, Michael Rath implemented the whole mining approach, designed the database layout, and performed the data collection. He also performed most of the writing. All authors then reviewed the paper prior to publication.

Paper II
Michael Rath was in charge for most parts of the paper. He came up with the mathematical background of the presented algorithm, the approach, and implemented the prototypical solution. Patrick Mäder and Michael Rath co-designed the study. The applied statistical methods were suggested by David Lo. He also provided a running version of the *AmaLgam* algorithm, which was used to compare localization performance as shown in the paper. Michael Rath wrote the majority of the paper and all authors reviewed the text.

Paper III
The idea for the paper were equally suggested by Jane Cleland-Huang and Patrick Mäder. As first author, Michael Rath was responsible for most of the work. He collected the data, implemented the majority of the approach, and performed all experiments. Jacob Rendall assisted Michael Rath during the implementation. The textual analysis in the paper was implemented and executed by Jin Guo. The user study was designed by Jane Cleland-Huang and conducted by her, Jacob Rendall, Jin Guo, and Patrick Mäder. Michael Rath wrote most parts of the paper, with contributions by Jane Cleland-Huang and Patrick Mäder. All authors reviewed the text.

Paper IV
Patrick Mäder proposed to create and publish this benchmark dataset. Michael Rath was the first author of the paper, created the required tools to collect the artifacts,

and performed the presented experiments. He also wrote most parts of the text, with contributions by Patrick Mäder who also reviewed the paper.

Paper V
Michael Rath developed the approach and the tool described in the paper. Mihaela Tomova assisted Michael Rath during the implementation. The majority of the text was written by Michael Rath with contributions of Mihaela Tomova who also created all illustrations. Patrick Mäder gave feedback and reviewed the paper prior publication.

1. Introduction

Developing software is a complex task, performed by humans, and defects are inevitable (W. E. Wong et al. 2016). It is costly, time-consuming, and requires extensive project knowledge to locate and remove (*fix*) these *bugs*. Studies estimated, that software testing and debugging consumes more than one third of the total cost of software development (Si, Hu, and Zhou 2010; Xie et al. 2014). A more recent research report states, that over 600 million hours are spent on debugging code in North America each year, which equates to $61 billion of salary costs alone (Cambridge Judge Business School MBA, Undo IO 2021). Further, the developers are often overwhelmed by the sheer amount of reported bugs. A developer at Mozilla is cited *"Everyday, almost 300 bugs appear that need triaging. This is far too much for Mozilla programmers to handle."* (Anvik, Hiew, and Gail C Murphy 2005). Similar numbers are found in other projects such as Eclipse, where the software team has to handle 115 bugs every day (Zhao et al. 2015).

The most time consuming task in bug fixing is to first locate the defective parts in the project's code base (Wang and Lo 2014). This is comparable to finding the needle in a haystack. Large projects consist of thousands of source code files, but only few of them are relevant for fixing an individual bug. A study of three projects and 374 bugs showed, that 88% of the bugs involve at most two source code files (Lucia et al. 2012). Additionally, the older a piece of software is, fewer developers know the whole code base. Manually locating bugs heavily relies on the developer's experience, judgement, and intuition to prioritize code to be correct or faulty (W. E. Wong et al. 2016). Especially young developers or those new to a project might struggle with these tasks. Thus it is highly desireable to automate the *bug localization* process to reduce the cost and the time spent. Two major approaches for bug localization emerged. The first is dynamically locating the bug via program execution by leveraging data monitoring techniques and breakpoints (Abreu et al. 2009). However, this dynamic approach is often time consuming and expensive itself (Saha et al. 2013). The second is locating the bug via different forms of static analysis using *bug reports* and the project's source code (Hovemeyer and Pugh 2004). This idea will be studied in detail in this thesis.

Companies and open-source projects use bug reports to collect bugs reported by developers, testers, and end-users to guarantee the quality of software. A bug report is a structured artifact containing information about the misbehavior. Standard

components of a bug report are environment information (product, version, operating system), management fields (reporter, assignee, priority), and arguable most import a free-text description (Herzig and Zeller 2014). This description characterizes the bug and hopefully provides steps to reproduce it. The description plays an important role for a specific class of static *bug localization algorithms.* These are based on *information retrieval (IR)* techniques and gained significant attention in the last decade. They have relatively low computational costs as it is not necessary to execute the software. Further, they have low dependencies only requiring the bug report and the project's source code (Manning, Raghavan, and Schütze 2008; Binkley and Lawrie 2010; W Bruce Croft, Metzler, and Strohman 2010). IR-based bug localization algorithms treat the source code files as text documents. The bug report's description is used as a query to search defective source code files. The result is a ranked list, based on predicted relevance, of candidate source code files that guide the responsible developer (assignee) of the bug report to fix the bug.

Software and systems *traceability* is widely accepted as an essential element for supporting many software development tasks such as change impact analysis, coverage analysis, and bug fixing (Cleland-Huang et al. 2014; Gotel et al. 2012). It is non-negotiable in development of safety-critical systems and thus part of standards ISO 26262 (International Organization for Standardization 2011) in the automotive industry and IEC 61511 (International Electronical Commission 2003) in the industry sector. Traceability documents relationships (*trace links*) between software artifacts. For example requirement-to-source code trace links may indicate where a specific requirement is implemented in the source code and vice versa. There is empirical evidence, that these trace links can accelerate bug fixes and feature extensions by $20-30\%$ and make 50% more correct (Mäder and Egyed 2011; Briand et al. 2014). But the effort needed to manually establish and maintain trace links is costly (Heindl and Biffl 2005), and has often been perceived as prohibitively high, especially in non-regulated domains. Thus, automatically creating requirements to source code trace links is studied by many researchers. IR-based algorithms are often used to solve the challenging task (Cleland-Huang et al. 2007; De Lucia, Fasano, and Oliveto 2008; Mahmoud and Niu 2010; Keenan et al. 2012; Rempel, Mäder, and Kuschke 2013).

Large amounts of development artifacts are created during the lifecycle of a project. It is not unusual that tens of thousand of them are involved (Feldt 2014; Regnell, Berntsson-Svensson, and Wnuk 2008) ans sometimes located in multiple software repositories. *Issue tracking systems* constitute prominent examples of software repositories utilized in commercial and open-source projects (Skerrett 2011). They are used to manage software requirements and bug reports, summarized under the umbrella term *issue* but also provide a communication hub for developers to ask for advice, and share opinions useful for maintenance activities or making decisions (Murgia et al. 2014). The continuous inflow of issues especially in projects applying

agile methodologies reflects the whole project's history. Another import software repository are *version control systems* storing the project's source code files and their evolution. Source code modifications are commonly subdivided into small, manageable increments (Kamei et al. 2013) and the version control system stores the transition from one version of a file to another (Hinsen, Läufer, and Thiruvathukal 2009).

The goal of this thesis is to fuse all introduced concepts, i.e., traceability information, and history of development artifacts available in software repositories to improve IR-based bug localization. Therefore, a process is presented that allows convenient access to a project's development artifacts. After analyzing the artifacts and trace links among them, two novel components for a bug localization are created that exploit this information. These components constitute the backbone for a new IR-based bug localization algorithm. Next, the ability to increase the existing issue to source code trace link set of a project is studied. This results in an automatic trace link augmentation algorithm derived from the project's development workflow. The developed algorithms are evaluated in five large empirical experiments. These are conducted on a dataset containing tens of thousands of artifacts.

The remainder of the thesis is structured as follows.

Section 2: Background. This section sets the context for this thesis and introduces fundamental concepts and ideas. It also provides references to relevant literature for further details.

Section 3: State of the Art. This section discusses the state of the art relevant for this thesis. It recaps bug localization algorithms, their internal structure, as well as the research area of trace link retrieval. The section ends with a critique, which identifies a number of shortcomings and thus motivates the goals of the thesis.

Section 4: A Holistic Approach to Improve IR-based Bug Localization. This section outlines the goals of this thesis and motivates novel ideas concerning IR-based bug localization. Also an artifact model and corresponding artifact relations is presented, which is used throughout this thesis.

Section 5: Mining Software Repositories to Create Holistic Datasets. This section introduces software repository mining. Project's artifacts, i.e. requirements and source code modifications, are usually scattered in multiple repositories. This section describes an approach to identify and collect (*mine*) a projects' repositories artifacts. The artifacts are stored within a database allowing easy access to analyze the retrieved data. The result is the *SEOSS* dataset containing 600,000 artifacts and 300,000 trace links among them.

Section 6: The ABLoTS Bug Localization Approach. At the beginning, this section recaps the internal structure and components of IR-based bug localization

algorithms. Than, the included similar bug report components are reviewed in detail. Afterwards, current shortcomings (e.g. limitation to only bug reports) are discussed and a novel approach called *TraceScore* is presented as core of a similar *issue* component. Further, the current approach for a structured source component is analyzed and enhanced leading to the *LuceneScore* approach. Finally, the previously obtained ideas are combined to create the novel bug localization algorithm *ABLoTS*.

Section 7: Automatically Augmenting Incomplete Issue-to-Commit Trace Links. This section addresses missing trace link in projects. Therefore a novel algorithm, *TLSA*, is presented to enhance and augment a project's trace link set. Its design is based on common developer workflows identified in software projects.

Section 8: Evaluation. This section evaluates all presented concepts and algorithms. It starts by stating six research questions. Next, three datasets are defined used throughout the evaluation. Several datasets are required, because the presented approaches require different information. After presenting the evaluation metrics, five experiments to answer the research questions are described in detail. Each experiment addresses a specific aspect within the holistic approach.

Section 9: Discussion. This section critically discusses the results of the five conducted experiments. Further, the current limitations of the applied approaches are discussed. The section ends with a detailed discussion of threats to validity for the conducted experiments. Each experiment poses different threats to validity and thus is handled separately.

Section 10: Conclusion and Future Work. This section concludes the thesis. It summarizes important findings and conclusions. Based on the findings, possible directions for future work are outlined.

Section A: Appendix. The appendix provides supporting material related to the evaluation and discussion.

2. Background

This chapter provides a brief overview of techniques, software engineering concepts, and ideas that are used throughout the thesis. It also provides references to relevant publications for further reading.

2.1. Defining Essential Terms

A *defect* is an error, which impairs the functions of a system. In context of software development defects are commonly termed as *bugs*. Bugs are introduced to the projects' source code because of mistakes made by developers. They stem from wrong implementations or software design flaws. Such a source code modification is called a *bug inducing* change (Wen, Wu, and Cheung 2016). If a defect is executed, it can lead to *failures*, i.e. the software no longer complies to its requirements. Thus, the program may produce incorrect or unexpected results, or behaves in unintended ways (Committee et al. 1990; Zeller 2009). Once a defect is discovered, a *bug report*[1] is filed. This software document, a special kind of software development *artifact*, describes the bug most commonly with natural language, created by developers, testers, or end-users (Zhang et al. 2015). For example, a bug report taken from project Hadoop[2] reads

> **Fix memory leak in `FileSystem.Cache.Key` class** Calling `FileSystem#get(final URI uri, final Configuration conf, final String user)` multiple times can result in memory leak because of the hash method implementation of `UserGroupInformation`. FileSystem always instantiates a new FileSystem object despite using the same user name/same URI. *[...]*

and the user reports a bug related to memory management. After reporting, the bug report is *triaged*. Thus the developers discuss whether a bug is a real bug, prioritize the bug report, and decide which developer becomes the *assignee* and should work on the bug report. *Bug localization* follows and the developer *triages* source code

1. Bug reports have many names, including "defect reports", "fault reports", and "failure reports" (Zhang et al. 2015). This thesis consistently uses the term "bug report".
2. https://issues.apache.org/jira/browse/HADOOP-13971

modifications to identify the bug-inducing one(s). Once identified, the process of *bug fixing* is performed, where the developer takes actions to remove the bug by creating *fix-inducing* source code modifications (Sliwerski, Zimmermann, and Zeller 2005). This removal is termed *debugging*, i.e., finding the root cause of the bug and remove it such that the failure no longer occurs (Zeller 2009). When a bug has been fixed, the corresponding bug report is marked as *resolved*.

2.2. Developing Software in an Agile Way

In the past decade, a number of notable strategic changes in industrial software engineering were made, including the adoption of *agile methodologies* (Highsmith and Cockburn 2001). Traditional software tried to anticipate a complete set of software requirements early in the development lifecycle and avoid change later to reduce cost. However, the increasing demand for acceleration of time to market, and customer satisfaction no longer suits the former approaches, but is addressed by agile methodologies. This thesis focuses on software development utilizing agile principles. A study shows that organizations with high agility meet business goals 19% more likely than those with low agility (Project Management Institute 2015). Moreover, the former finish projects on time 65% of the time, versus 40% for the latter. Agile methods provide a new set of 12 principles formulated in the *agile manifesto* (Agile Manifesto Team 2001). This includes *"welcome changing requirements, even late in development"*, *"deliver working software frequently"*, *"build projects around motivated individuals"*, and *"working software is the primary measure of progress"*. Thus, agile development no longer follows a plan, which usually outdates quickly. Instead short iterative cycles, the *sprints*, with feature planning, dynamic prioritization, implementation, and testing are established (Highsmith and Cockburn 2001). Instead of planing tasks, the focus is on features, i.e. the things customers understand and care about. Open-source software (OSS) development and agile share a lot of underlying principles such as skilled individuals, self-organizing teams, short feedback loops, embraced change, and frequent releases of working code (Koch 2004).

Current software systems are continuously changing with regard to features and source code changes and thus require effective ways to track and relate all modifications. This is solved by introducing navigable *trace links*. Many studies have shown that established traceability is beneficial for developing well engineered software (Gotel and Finkelstein 1994; Gotel et al. 2012; Cleland-Huang et al. 2014; Mäder and Cleland-Huang 2015). The next Section gives a brief introduction of the underlying concepts (see Sec. 2.3).

Each sprint provides a small increment of the software. Thus large features are split into smaller ones, and developers realize them usually in a sequence of reasonable source code modifications (Kamei et al. 2013), in order to avoid breaking the software (i.e. inability to create a deliverable or introducing a major bug). Thus sprints produce a stream of artifacts, such as new features, source code modifications, tests, and bug reports. A new set of tooling is required to manage the artifacts and implement an agile software development process. This set needs to support feature management, agile boards for feature planning, custom workflows, and handling continuous source code changes. Further, the tools need to scale, should be decentralized, and support effective ways for collaboration and communication among developers. This is especially import in OSS development, because of the hundreds or thousands of volunteer developers working from all around the globe and rarely meet in person (Mockus, Fielding, and Herbsleb 2000). Notable tools in this regard are discussed in Section 2.4.

Embracing change first seems to contradict with retaining quality. Customers demand and expect high quality, innovative software delivered as soon as possible (Highsmith and Cockburn 2001). The agile response includes constantly testing the software, for earlier and less expensive bug detection. Once a bug has been detected, the bug localization process starts in order to identify the defective source portion. Automating this task is highly beneficial for agile approaches, because it accelerates testing and reduces cost.

2.3. Software Traceability

Artifacts are continuously created and modified when agile methods are used for software development. Therefore it is vital to track and relate all these activities. Traceability represents the *"potential to relate data that is stored within artifacts of some kind, along with the ability to examine this relationship"* (Gotel et al. 2012). Navigable links need to be created to connect the data held in the artifacts in order to achieve traceability. Once these links are established, traceability provides support for many different software engineering activities including change impact analysis, test regression selection, safety analysis, and coverage analysis (Gotel and Finkelstein 1994; Cleland-Huang et al. 2014).

This thesis follows the definition of software traceability as proposed by the *Center of Excellence for Software & Systems Traceability (CoEST)* (Gotel and Finkelstein 1994; Center of Excellence for Software & Systems Traceability 2020):

> **Software traceability** is the ability to describe and follow the life of any uniquely identifiable software engineering artifact to any other in

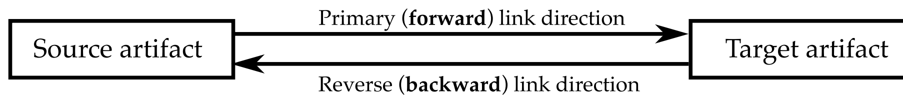


Figure 2.1.: Definition of trace link directionality. Typically only the forward direction is explicitly defined, but technically a reverse (backward) link is also created.

> both a forwards and backwards direction.

Using this definition, two main building blocks of traceability can be identified: *artifact* and *trace links*. Gotel et al. (Gotel et al. 2012) define these terms as follows.

> A **(trace) artifact** is a traceable unit of data and is qualified as either a *source artifact* or as a *target artifact* when it participates in a *trace*.
>
> A **trace link** is a specified *association* between a pair of artifacts, one comprising the source artifact and one comprising the target artifact.
>
> A **trace** (noun) is a specified triplet of elements comprising: a source artifact, a target artifact and a trace link associating the two artifacts.

The terms source and target artifact imply directionality for trace links. However, in practice every trace link can be traversed in both directions (Gotel et al. 2012) as depicted in Figure 2.1. All traces in a project are termed as *trace set*, and can be represented as a traceability graph (Center of Excellence for Software & Systems Traceability 2020).

> A **traceability graph** is a representation of the trace set, i.e. all traces of a project, with trace artifacts depicted as nodes and trace links depicted as edges.

A path in the traceability graph from a source artifact to a target artifact following the edges (i.e. using trace links) is called **trace path**.

2.4. Common Tools Utilized in the Agile Development Process

This section introduces common tools and software repositories used in agile software development. Nowadays, two software repositories are very common: *issue tracking systems (ITS)* and *version control systems (VCS)*. Depending on the software project's

size and internal structure it may utilize multiple ITS and VCS. In this thesis, *project data* denotes the artifacts, their evolution (history), and relationships among them stored in ITS and VCS.

2.4.1. Managing Requirements and Bug Reports in Issue Tracking Systems

The predecessor of issue tracking systems were *bug tracking systems*, which solely focused on bug reports and their management. In open-source projects these systems are an important part of how teams (such as *eclipse* and *mozilla*) interact with their user communities (Breu et al. 2010). A bug tracking system allows the users to enter bug reports and keep track of their status'. Over the years, these systems were generalized to issue tracking systems, which also allow to manage additional artifacts including requirements, software features, and open tasks (Skerrett 2011). Developers can choose from different ITS implementations, both commercial and open-source ones (Singh and Chaturvedi 2011; Project Management Zone 2018). Popular implementations are *BugZilla*, *Mantis*, *Trac*, and *Atlassian Jira*.

Out of them, the commercial ITS Atlassian Jira (Atlassian Corporation 2020b) is of special interest for this thesis. It is widely used[3], and is also very popular in the open-source community, because of its "friendly" licensing scheme[4]. Jira consists of multiple modules. The *Jira Software* module supports agile project management and issue tracking. Jira is highly customizable and can be adapted to project specific needs, e.g. defining artifact types, their properties, and workflows. Customizations are usually applied, and therefore it is difficult to make universally valid statements how Jira is used for a given project. However, the basic installation comes with a set of defaults, which are present in most project.

The fundamental artifact in Jira is called *issue*. Figure 2.2 shows the issues view, i.e. the list of most recent issues, of project Derby[5]. Derby is a relational database management system developed by the Apache Software Foundation. The issue overview allows to browse, filter, and search the project's issues. The most important properties of an issue are shown as columns, and selecting an issue reveals all details.

3. Atlassian claims about 10 million daily users. https://www.atlassian.com/customers
4. https://www.atlassian.com/software/views/open-source-license-request
5. With respect to developers' privacy, all personal information like names and pictures is obfuscated (grey bars) throughout this thesis.

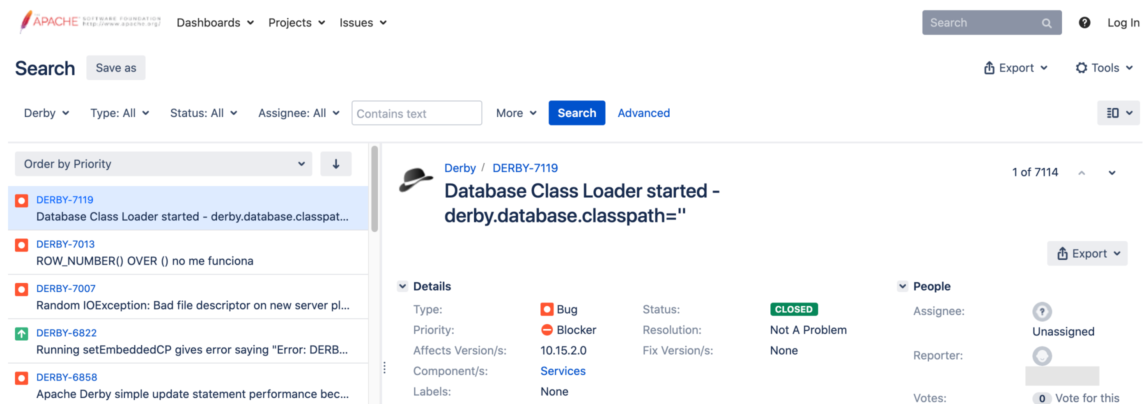


Figure 2.2.: Screenshot of issue overview of project Derby in Atlassian Jira. It shows a list of the most important properties such as summary, status, and resolution of the most recent issues.

Properties of Issues in Atlassian Jira

Each issue in Jira is uniquely identifiable by its *issue id* derived from the project name and an increasing positive integer number. For example `DERBY-6453`[6] is a valid issue id in project Derby and the typed properties of this issue are shown in Figure 2.3.

This section describes the properties that are relevant in this thesis. Most properties and their values are customizable, and depending on the project new properties can be created. The default property list includes the texts *summary*, a brief description of the issue, as well as a longer *description*. Further, there are properties to capture stakeholder information about who reported the issue (*Reporter*) and who is responsible to solve it (*Assignee*), as well as information recording timestamps about issue creation and modification (*Created*, *Updated*, *Resolved*). Important issue properties, their values, and the suggested interpretation are shown in Table 2.1 (Atlassian Corporation 2021a). The issue's *type* is used to distinguish types of work in unique ways, and help to identify, categorize, and report on the stakeholders work. The suggested issue types include *Bug*, *New Feature*, and *Task*, but this list and the intended usage has changed between different Jira releases (Atlassian Corporation 2021b). Older Jira versions also contained the type "Improvement", which is an improvement or enhancement to an existing feature or task. Figure 2.4 shows the distribution of used issue types found in 33 projects, which will become of interest in Section 5. The x-axis depicts the project, and the y-axis all found issue types. The color of the dots is the percentage of issues with this type in the respective project. The issue types "Task", "Sub-task", and "Bug" are present in every project, whereas

6. https://issues.apache.org/jira/browse/DERBY-6453

Figure 2.3.: Screenshot of improvement `DERBY-6453` of project Derby.

"Bug" has the most occurrences. The second most prominent issue types are "New Feature", "Improvement", "Feature Request", and "Enhancement". Interestingly, there seems to be a usage pattern such that only two of them are used in a project. Because of customization, some obscure issue types also exist, such as "Component Upgrade" in project Wildfly. In Figure 2.3, the issue has type "Improvement" encoded as green icon in front of the issue identifier `DERBY-6453`. In this thesis, the following mapping is used.

> The term **bug report** maps to the Jira issue type "Bug", because this allows no misinterpretation and the shown example projects agree in that respect.
>
> All issue types different from "Bug" are summarized as **requirement**s. As shown, most projects contain in addition only "New Feature", "Feature Request", "Improvements", "Enhancements", and "Tasks". They all represent a natural language description of necessary changes to be made in the source code. This matches the IEEE definition for a requirement: *"A condition or capability needed by a user to solve a problem or achieve an objective."* (Committee et al. 1990).

For example, the summary of improvement `DERBY-6453` clearly refers to source code elements, such as class name `InsertResultSet`, or identifier `skipCheckConstraints`. The chosen mapping will be formalized in Section 4.2.

In Jira, the project's workflow is used to track the lifecycle of an issue. The workflow is a record of statuses and transitions of an issue during its lifecycle. An issue's status indicates its current place in the project's workflow. The workflow itself is

(a) Issue type

Issue type	Description
Bug	A problem that impairs or prevents the functions of the product.
New Feature	Requesting new capability or software feature.
Task	A task represents work that needs to be done.

(b) Issue status

Issue status	Description
OPEN	The issue is open and ready for the assignee to start work on it.
IN PROGRESS	This issue is being actively worked on at the moment by the assignee.
RESOLVED	A resolution has been taken, and it is awaiting verification by reporter. From here, issues are either reopened, or are closed.
CLOSED	The issue is considered finished. The resolution is correct. Issues which are closed can be reopened.
REOPENED	This issue was resolved previously, but the resolution was either incorrect or missed a few things or some modifications are required.

(c) Issue resolution

Issue resolution	Description
DUPLICATE	The problem is a duplicate of an existing issue.
DONE	Work has been completed on this issue.
FIXED	The issue has been fixed, i.e. successfully resolved.
WON'T FIX	Work on the issue is rejected.

Table 2.1.: Important issue properties, their values, and suggested interpretation in Atlassian Jira (Atlassian Corporation 2021a).

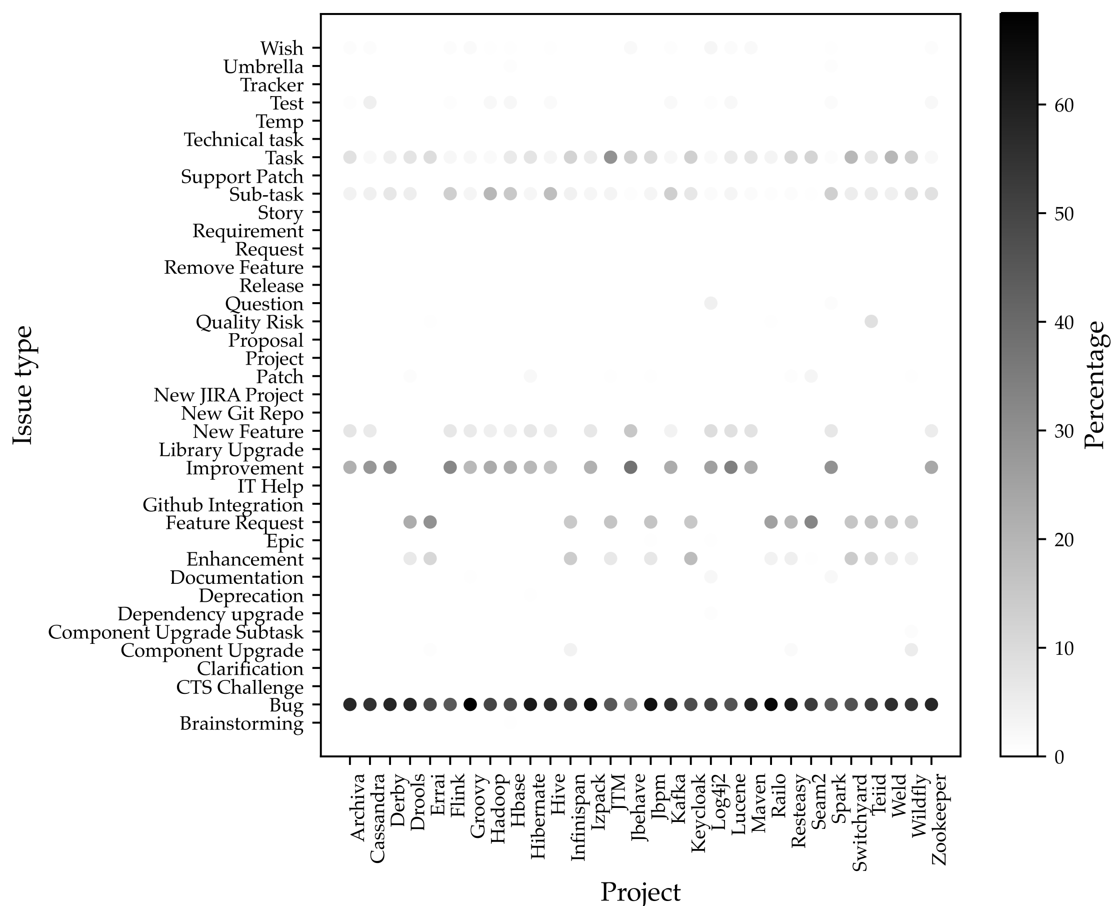


Figure 2.4.: Issue type distribution in 33 example open-source projects (Rath and Mäder 2019b). The type "Bug" is present in every project, and the majority issues of a project have this type.

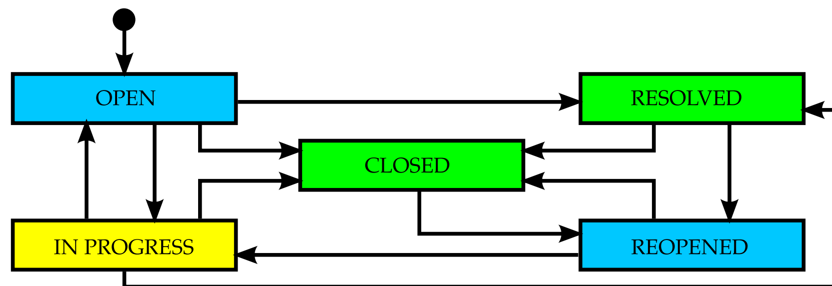


Figure 2.5.: Example issue workflow schema in Atlassian Jira. Blue represents states, where no-one is working on the issue, yellow are intermediate states, and lastly green represents final states.

represented by a schema, such as the one depicted in Figure 2.5. The transitions are represented as links between the statuses. A transition is necessary to move the issue from one status to the next one. An issue can be only in one status at any given point in time. In the shown workflow an issue starts its lifecycle with status OPEN. The blue highlighted states represent initial ones, where no-one is working on the issue. Actively working on the issue is denoted by state IN PROGRESS. Eventually the work on the issue has finished and a final state (highlighted in green), is reached and the issue's resolution is set. Depending on the project, FIXED is used as resolution if a bug was successfully removed, and in case the issue type is not "Bug", DONE is used. However, e.g. project Railo also uses DONE as bug report resolution. The example issue `DERBY-6453` has the status CLOSED with resolution FIXED (see Fig. 2.3).

Atlassian Jira also supports associations between issues called "Linked issues". These typed links such as "duplicates", "relates to", or "blocks" allow to establish trace links among issue artifacts. For example, issue `DERBY-6453` traces to improvement `DERBY-532` using the trace link type "relates to".

2.4.2. Managing Source Code Files in Version Control Systems

A *version control system (VCS)* allows to systematically capture the evolution of files, especially source code files. The VCS allows the developers to coordinate a collaborative software development. Thereby, the VCS stores the individual versions of a file and also information about the applied modification, i.e. the transition from one version to another version (Hinsen, Läufer, and Thiruvathukal 2009). Like for

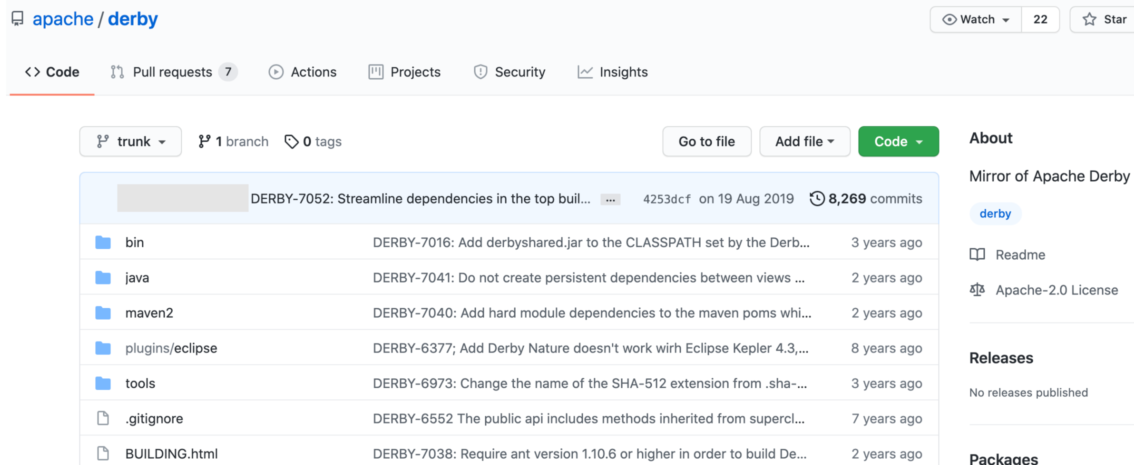


Figure 2.6.: Screenshot of project Derby in GitHub (Github, Inc. 2021).

ITS, many VCS implementations are available, such as *Git*, *mercurial*, or *SubVersion*. During the last years, Git (Git Community 2020) has become the dominant choice for version control, and thus is the only one considered in this thesis (Stackoverflow 2018). Figure 2.6 shows the main view of project Derby in *GitHub* (Github, Inc. 2021). GitHub is a popular hosting website for software projects that use Git. It extends the capabilities of git by offering additional features, such as extended collaboration features, continuous integration, and wiki pages. From the depicted main view in Github, users can browse the project's files, their history, and the information about all changes, i.e. the individual *commit* artifacts.

Properties of Commits in Git

The version control system Git (Git Community 2020) uses atomic *commits* to represent changesets of files, i.e. the edits required to go from one version of a file to the next version. A commit is able to hold changesets of multiple files. Each commit is uniquely identified by a 160bit SHA-1 hash value. For example, the hash value of the commit shown in Figure 2.7 starts with `fb69172` and contains modifications of the source code file `TableDescriptor.java`. Specifically the comment in line 734 is removed. Additional properties are available for a commit. They include the stakeholder who created the commit, the creation timestamp, and a textual commit message stating the purpose of the change, e.g. *"Remove dead code in [...]"* in the depicted commit.

DERBY-6453 Remove dead code in InsertResultSet and flag skipCheckCons… Browse files

```
…traints

Patch _*cleanup-misc-3*_ which removes dead code but inserts asserts
in sane mode if we should somehow end up with triggers in bulk insert
mode and also removes a boolean variable in some interfaces which was
always called with false: {{skipCheckConstraints}}.
It adds a new test case, {{CheckConstraintTest#testbulkInsert}}.


git-svn-id: https://svn.apache.org/repos/asf/db/derby/code/trunk@1556938 13f79535-47bb-0310-9956-ffa450edef68
```

trunk

committed on 9 Jan 2014 — 1 parent 584c0fb — commit f16776261c331f4b37a4e702a7be2d6539e571a7

Showing 8 changed files with 99 additions and 156 deletions. Unified Split

```
8 java/engine/org/apache/derby/iapi/sql/dictionary/TableDescriptor.java
@@ -731,7 +731,6 @@ public int getQualifiedNumberOfIndexes(int minColCount,
731 731      *     statement type and its list of updated columns.
732 732      *
733 733      *     @param  statementType              As defined in StatementType.
734     -    * @param     skipCheckConstraints   Skip check constraints
735 734      * @param     changedColumnIds           If null, all columns being changed, otherwise array
736 735      *                                               of 1-based column ids for columns being changed
737 736      *     @param  needsDeferredProcessing IN/OUT. true if the statement already needs
@@ -745,7 +744,6 @@ public int getQualifiedNumberOfIndexes(int minColCount,
745 744    public  void    getAllRelevantConstraints
746 745    (
747 746        int                                   statementType,
```

Figure 2.7.: Screenshot of commit `f167762` of project Derby in GitHub.

2.4.3. Creating Trace Links between Artifacts in Jira and Git

Atlassian Jira and Git are separate, unconnected software systems, but often used together in agile software development. The artifacts representing source code changes are maintained in Git, and the issue artifacts in Jira. However, there is the need to interconnect the systems. For example, developers create requirement artifacts in Jira and start making changes to the source code files accordingly. Without tracing from the issues to commits, the knowledge why these source code modifications were made is lost. Thus it has become a common practice for developers to *tag* commit messages with issue id(s), i.e. mention the issue id typically at the beginning of the commit message (Cubranic and Murphy 2003; Fischer, Pinzger, and Gall 2003).

This procedure is even reflected in the guidelines of the Apache Software Foundation stating that *“You need to make sure that the commit message contains at least [...] a reference to the Bugzilla or Jira issue [...]”* (Apache Software Foundation 2020). It is also part of the developer documentation of projects like in Hadoop (*“The commit message should include the JIRA issue id [...]”* (Hadoop Community 2020)), or in project Subversion (*“If your change is related to a specific issue in the issue tracker, then include a string like ‘issue #N’ in the log message [...]”* (Subversion Community 2021)).

This tagging procedure allows to connect the former independent systems Jira and

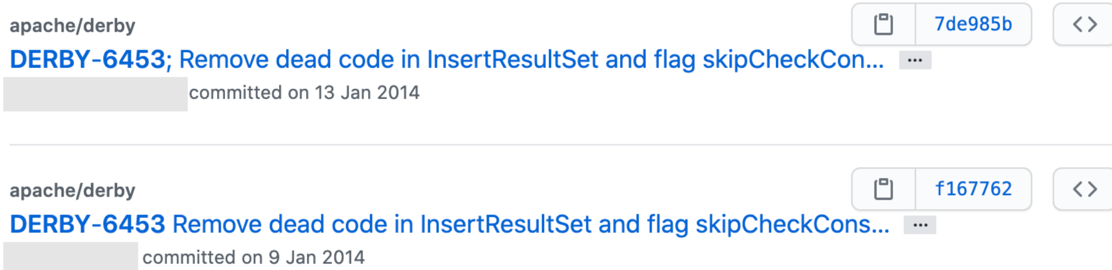


Figure 2.8.: Screenshot of two tagged commits to create trace links to Jira issue with identifier `DERBY-6453`.

Git and provides an important part in the creation of project-wide traceability. Figure 2.8 shows an overview of two commits of project Derby, including the one with commit hash `f167662` (see Fig. 2.7). Each commit message starts with issue id `DERBY-6453` and thus creates a trace link to the improvement shown in Figure 2.3.

Nowadays developers actively utilize the trace links in their daily work. For example, several browser extensions exist to integrate GitHub with Jira by replacing the issue identifiers added to commit messages with hyperlinks which allows to navigate between both tools (Sullivan 2014; D'Haeseleer 2019). Eventually GitHub itself added support for this functionality in 2019 (GitHub, Inc. 2019). The introduction of smart commits further extends the capabilities beyond navigation (Atlassian Corporation 2020a). These allow to process Jira issues using special commands and enable commenting on issues, record time tracking information, or transition issues to any status defined in Jira project's workflow (see Fig. 2.5).

Combining the information of the presented examples allows to create a (partial) traceability graph for the improvement `DERBY-6453` as shown in Figure 2.9. This improvement relates to issue `DERBY-532`, which is modelled in Jira. Two commits, `f167762` and `7de985b`, tag issue `DERBY-6453`. The tagging procedure allows to trace between the independent systems Jira and Git. In Git, each commit inherently contains (traces to) modifications of one or more files.

2.5. Localizing Bugs using Information Retrieval Techniques

ITS such as Atlassian Jira allow to store and manage different types of issues, including bug reports. Usually these reports are written in natural language, having a short title and a longer bug description. It provides information about the abnormal

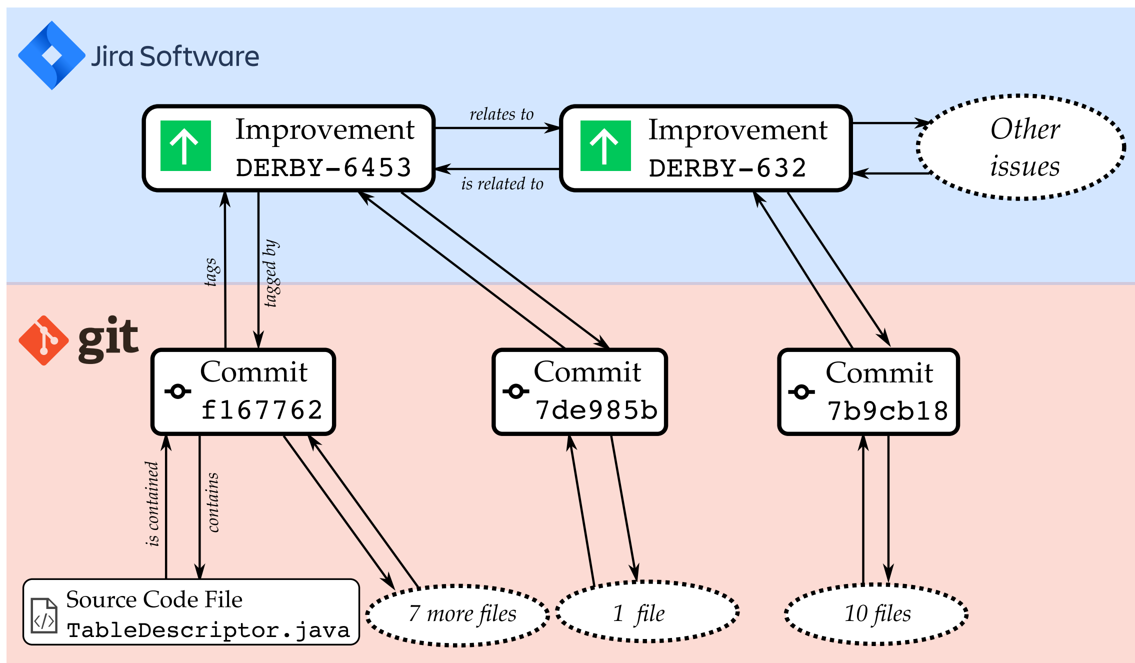


Figure 2.9.: Example for a traceability graph based on the examples presented in this section. Atlassian Jira directly supports to trace between issues. In Git, the commits inherently contain modifications of source code files. Tagging the commit messages with Jira issue identifiers allows to trace between Jira and Git.

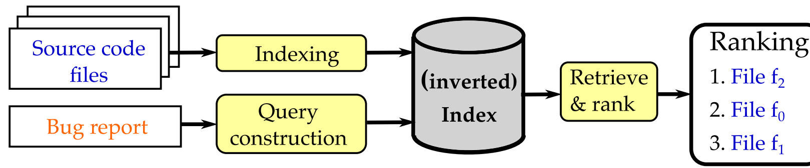


Figure 2.10.: Structure of an IR-based bug localization algorithm.

behavior and initially guides the developer in retrieving source code files to be modified for removing the defect. However, the quality of bug reports differs widely (Zimmermann et al. 2010). As introduced, in Atlassian Jira bug reports are issues of type "Bug" having the same properties as other types and follow the same project's workflow.

Bug localization is a task in software maintenance, in which a developer uses information about a bug present in a software system to locate the portion of source code that must be modified to remove the defect (see Sec. 2.1). If done manually, this process is prone to errors. Further, the developers need to bridge the *lexical gap* between the description written in natural language and the formal representation of the source code. This requires extensive project knowledge and often the bug reports do not contain the needed information (Zimmermann et al. 2010; Breu et al. 2010). A *bug localization algorithm* automates the localization process.

A common realization of bug localization algorithms is leveraging *information retrieval (IR)* techniques (Zhou, Zhang, and Lo 2012; Gay et al. 2009; Lukins, Kraft, and Etzkorn 2008; Rao and Kak 2011). Manning et al. define information retrieval as *"[...] finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."* (Manning, Raghavan, and Schütze 2008). In the context of bug localization this translates to the following. The material to find are *relevant* source code files, the information need is to locate the bug, and the large collection is the project's code base. Both the bug report and the source code files are treated as (text) documents, and the bug report is used to *query* and rank the source code files. The query is what the developer conveys to the computer to express the information need. The collection of source code files is termed *corpus*: a body of text. A returned source code file is relevant if it contains the information the developer was looking for, i.e. the bug.

The whole process can be represented by three steps (Zhou, Zhang, and Lo 2012) as visualized in Figure 2.10.

1. The first step handles lexical analysis of the project's source code files and creates the corpus. The corpus is **indexed** and results in an (inverted) index.

2. The second step **constructs** a **query** from the bug report.
3. At last, relevant source code files are **retrieved** and **ranked**.

The details of the steps are presented in the next sections.

2.5.1. Indexing the Source Code File Corpus

This step performs lexical analysis and prepares the source code corpus for indexing. Therefore a list of preprocessing steps is applied as shown in Figure 2.11. This starts with *tokenization*, i.e. the textual content is chopped up into pieces called *tokens*, and perhaps throwing away certain characters. Next, stopword removal is applied i.e. the tokens are filtered by common (English) words (e.g. “a”, “in”, “the”), because they appear very frequently in documents and thus have less discriminating value. In context of source code this step also removes programming language keywords. Indentifiers and types in source code are often combined words concatenated using *camel case* (e.g. `runLexicalAnalysis`) or *snake case* (e.g. `run_lexical_analysis`). These compound words are split. Next, the tokens are converted to lowercase and *stemmed*. Stemming returns the stem for a token, i.e. by removing inflection, plural forms, and so on. For example the tokens `compute`, `computer`, and `computing` have all the same stem `compute`. The porter stemmer is a popular stemming algorithm used for this activity (Porter 2006). Corpus creation treats the source code files as *bag-of-words*, and thus only contained tokens are relevant independent of their ordering. From now the tokens are called *terms* which are finally used in the IR system. The preprocessing steps highly depend on the input documents, and thus individual steps may be skipped (e.g. no lowercasing), repeatedly applied (e.g. filtering again after compound word splitting), reordered, or new ones added (e.g. depending on programming language).

The set of unique terms represents the corpus’ *vocabulary*. After lexical analysis the source code files are indexed. This creates a mapping from a term to the list of source code file(s) the term occurs in. The mapping forms the *inverted index*. Here “inverted” is actually redundant, because an index always maps back, but the name has become the standard term in information retrieval (Manning, Raghavan, and Schütze 2008).

2.5.2. Constructing a Query from a Bug Report

In bug localization the bug report is used as a query to search for relevant source code files. Lexical analysis is applied to the bug reports texts, i.e. summary and description, to create the query. The same preprocessing steps as used for corpus creation can be used, or adapted to specific needs.

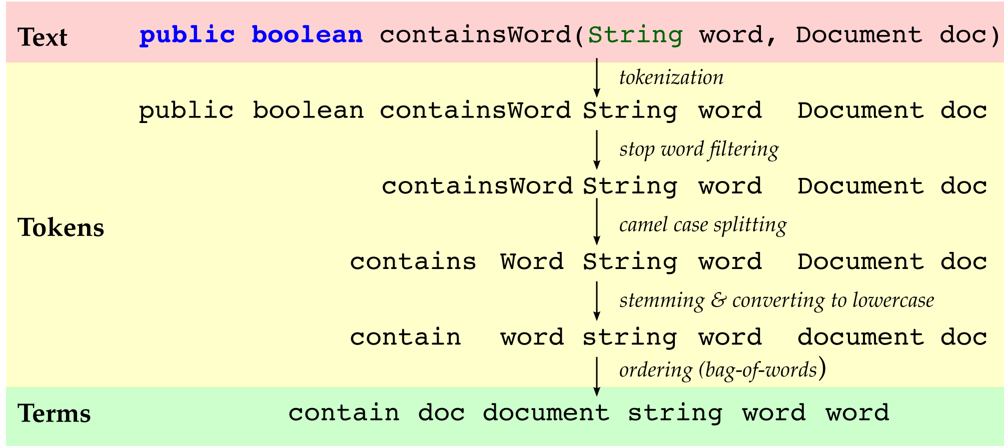


Figure 2.11.: Example of possible preprocessing steps applied during lexical analysis of source code files. It starts with input text: a method signature in Java programming language in this example. The source code text is transformed to tokens and finally terms are created, which are used by the IR system.

2.5.3. Retrieve and Rank Source Code Files

In this last step, a similarity score between the query and each document in the corpus is calculated and afterwards used to rank the source code files. Various approaches for this calculation exist (Dit et al. 2013), and a common one uses a *Vector Space Model (VSM)* (Manning, Raghavan, and Schütze 2008; Rao and Kak 2011). Here, the query q and the source code files (documents d) of corpus D, i.e. $d \in D$, are represented as weighted vectors $V_q, V_d \in \mathbb{R}^{1\times n}$

$$V_d = (w_{t_0,d}, w_{t_1,d}, \dots, w_{t_{n-1},d})$$
$$V_q = (w_{t_0,q}, w_{t_1,q}, \dots, w_{t_{n-1},q})$$

Each dimension $n = |\mathrm{Voc}|$ in the vector corresponds to a specific term in the vocabulary $t_i \in \mathrm{Voc}$. As already mentiond, this bag-of-words model just describes word occurrences while ignoring the relative position of the words in the document. Using the vectors, a similarity score sim between the query and each source code file is calculated using *cosine similarity* (see Eq. 2.1)

$$\mathrm{sim}(d,q) = \frac{V_d \cdot V_q^T}{\|V_d\| \cdot \|V_q\|} \tag{2.1}$$

Cosine similarity caculates the radian angle between the query vector and the respective source code file vector. The result is in range $[0 \dots 1]$, where 1 is the maximum similarity of the vectors and thus the documents are identical, and 0 is maximum dissimilarity. The vector weights $w_{t,d}$ for each term $t \in \text{Voc}$ are determined using the *term frequency (tf)* and *inverse document frequency (idf)*. Here tf specifies how often the term occurs in a document, and idf the number of documents containing t. Over the years, multiple *tf-idf* calculation schemes have been proposed, whereas the logarithmic variant achieves good results (Dumais 1991; W. Bruce Croft, Metzler, and Strohman 2009)

$$\begin{aligned}
f_{t,d} &= \text{frequency of term } t \text{ in document } d \\
\text{tf}(t,d) &= \log(f_{t,d}) + 1 \\
\text{idf}(t,D) &= \log \frac{|D|}{|\{d \in D | t \in d\}|} \\
w_{t,d} &= \text{tf}(t,d) \cdot \text{idf}(t,D)
\end{aligned} \tag{2.2}$$

The weight $w_{t,d}$ is highest when t occurs many times in a small source code file, lowest when t occurs in all source code files. Values between these edge cases are that t occurs fewer times in a source code file, or occurs in many source code files.

Finally, after calculating sim for all source code files and the query, the files are ranked according to this value. This results in a list with the source code file having the highest similarity on top. In Figure 2.10 the ranking is f_2, f_0, f_1. So the developers are adviced to look at these source code files in that order to locate the bug.

Besides its rather simplistic idea, the vector space model is still the backbone of many algorithms used today.

3. State of the Art

This thesis investigates bug localization in a project wide, holistic approach. Therefore, at first an overview on the existing body of work on IR-based bug localization is given (see Sec. 3.1). Related problems and techniques are found in the domain of *trace link recovery*. Here the goal is to automatically establish trace links among artifacts, to support tasks like change impact analysis, and coverage analysis. Section 3.2 discusses the state of the art in this domain.

3.1. Bug Localization

A common approach in automated bug localization is formulating the search process as an information retrieval problem (Manning, Raghavan, and Schütze 2008; Rao and Kak 2011; Gay et al. 2009; Lukins, Kraft, and Etzkorn 2008). Here, the bug report is treated as one textual document and used to query a list of documents. The projects' source code repository constitutes the query list in this context. The result is a ranked list of most relevant matches among all source code files. Now, a developer inspects the ranked list and eventually can fix the bug. Such methods are called information-retrieval-based bug location methods and do not require to execute a program, which is necessary in *spectrum-based* fault localization techniques (Abreu et al. 2009; Jones and Harrold 2005; Liu et al. 2006). A user study has shown, that developers benefit from high-quality bug reports as much as from IR-based algorithms (Wang, Parnin, and Orso 2015). Even then, the authors argue, that this only helps to quickly locate the defective source code files but does not help the developers to solve the most time consuming task, i.e., fixing the bug.

Multiple techniques have been proposed to improve the matching performance of IR-based methods (Saha et al. 2013; Wang and Lo 2014; Moreno et al. 2014; Wang and Lo 2016; Ye, Bunescu, and Liu 2014; Lukins, Kraft, and Etzkorn 2008; Marcus and Maletic 2003; Zhou, Zhang, and Lo 2012). The differences mainly focus on the retrieval and ranking of results.

A basic idea is the *Vector Space Model (VSM Sec. 2.5.3)*, which is utilized in many approaches (Zhou, Zhang, and Lo 2012; Wang and Lo 2014, 2016). When using VSM, the documents (bug reports and source code files) are encoded as vectors of token weights. The weights are usually computed using the token frequency (tf), i.e.,

how often a term occurs in a specific document, and inverse document frequency (idf), i.e. how often a term occurs in all documents (Manning, Raghavan, and Schütze 2008). The cosine similarity between two vectors, i.e. the representation of a source code file and that of the bug report, reflects the relationship of these artifacts. A value of 1.0 reflects a perfect match and 0.0 is maximal dissimilarity.

Latent Semantic Indexing (LSI) extends the vector space model (Marcus and Maletic 2003; Deerwester et al. 1988). This method can identify the relationship between the terms and concepts contained in an unstructured collection of text. This is especially import if a (short) query and (longer) documents do not share the same terms, but similar concepts like “car” and “automobile”. The core idea is to collect information about the context where a particular word appears, or not, and than to derive a set of mutual constraints that determine the similarity of sets of words to each other. The mathematical foundation of LSI is singular value decomposition (SVD). After constructing a term-by-document matrix from the user corpus, SVD is applied to create an LSI subspace. For example, in this subspace the original words “car” and “automobile” would map to the same concept and thus provide a solution to the synonym problem. New document and query vectors are orthogonally projected onto the LSI subspace. Cosine similarity is used to calculate the of the resulting vectors. Marcus et al. applied LSI to trace from documentation to source code (Marcus and Maletic 2003). In (Poshyvanyk et al. 2006), the authors used LSI for bug localization.

The *Latent Dirichlet Allocation (LDA)* is a topic modeling technique based on a statistical model. Each document is viewed as a mixture of multiple (latent) topics. A topic is a probability distribution based on frequently co-occurring words over the vocabulary of a document collection. The words in a topic are often semantically related and thus provide meaning to the abstract concept of a topic. For example, the words “human”, “genome” and “dna” might be the words with highest probability in a topic about “genetics”, because these words often appear together in this context. LDA possibly assigns multiple topics to a document indicating that the text is related to the topic “genetics” and “biology”. Thus, after applying LDA to a collection of documents it is possible to cluster them based on topics and to retrieve all documents related to “biology”. In bug localization the documents are source code files and bug reports. The similarity between them is calculated as conditional probability. LDA is successfully used by Lukins et al. (Lukins, Kraft, and Etzkorn 2008). In two case studies the authors show, that LDA based bug localization outperforms LSI methods. In (Nguyen et al. 2011), the LDA model is revised by using two topic models, one for bug reports and the other for source code files, which are than connected. The rational behind this design is, that technical topics must be mentioned in a bug report. Compared to the approach of Lukins et al., this architecture improves the bug localization performance. However, a comprehensive study by Roa et al. showed,

that sophisticated models like LSI and LDA do not outperform much simpler models like VSM (Rao and Kak 2011).

Researchers also introduced machine learning models, such as *learning to rank* (Ye, Bunescu, and Liu 2014) or *neural networks* (Lam et al. 2015) to address bug localization. Here, the models' free parameters (e.g. number of topics K for LDA) are no longer manually assigned and instead learned from training data. Further, machine learning approaches also replace the bag-of-words model by using word embeddings, and thus also incorporate the token ordering (token context) of the documents (Lam et al. 2015; Ye et al. 2016). The authors show an improved localization performance on their respective datasets.

Next to advances in natural language techniques to process bug reports and source code files, researchers also started to further analyze bug reports. Thus, the bug reports and source code files are no longer treated as a "just" texts and instead are seen as structured information (Saha et al. 2013). In (Saha et al. 2013) the authors construct *abstract syntax trees* from the source code files to extract different structural elements (i.e. comments, method names) separately. Also, the bug report title and description are no longer combined and handled independently. The extracted parts are used as individual queries, resulting in separate search scores, which are finally combined using summation. In the evaluation, their structured approach BLUiR outperforms BugLocator (Zhou, Zhang, and Lo 2012), which applies an unstructured bug report handling using VSM. Schröter et al. focused on stack traces found in bug reports (Schröter, Bettenburg, and Premraj 2010). The hypothesis is, that class and method names found in stack traces helps to find defective code locations. The author's evaluation shows, that developers benefit from this information and stack trace in bug reports speed up the localization process. Therefore, bug localization algorithms were designed to leverage stack traces as well (Moreno et al. 2014). However, the authors only provide theoretical results by adjusting parameters in case the developed algorithm is applied to bug reports not containing stack traces.

The *lexical gap* between queries (bug reports) and source code files usually limits the performance of IR-based models (McMillan et al. 2012). Thus, additional information and project-specific resources are utilized as well. This led to bug localization algorithms containing an ensemble of components, each calculating a separate ranking of source code files. These rankings are aggregated to the final ranking result, i.e. the overall output of the bug localization algorithm. Multiple aggregation methods were proposed, including summation (Saha et al. 2013), empirically determined weighting schemes (Zhou, Zhang, and Lo 2012; Wang and Lo 2014), trained weights using support vector machines (Ye, Bunescu, and Liu 2014), or even neural networks (Lam et al. 2015). Additional textual resources, e.g. API documentation, were used to improve localization results (Lam et al. 2015; Ye, Bunescu, and Liu 2014; Ye et al. 2016). Further, in (Lam et al. 2015) the authors utilize developer names and

project release information. Next to the (structured) text matching component, previously solved bug reports (project version history component) are used as another input for the bug localization algorithm (Zhou, Zhang, and Lo 2012; Saha et al. 2013; Wang and Lo 2014, 2016; Lam et al. 2015). The reasoning is, if the current bug report at hand is similar to a previously fixed one, the same source code files may need to be modified to resolve the current bug report.

Instead of processing the projects' source code files as a whole, Wong et al. (C. Wong et al. 2014) dissect each file in equally sized chunks. Afterwards the chunk with the highest similarity to the bug report is chosen to represent the file.

Wen et al. (Wen, Wu, and Cheung 2016) questioned the usefulness of a ranked list of source code files, because of the granularity of the results. Thus they designed the IR-based algorithm Locus to locate bugs from source code changes, which have a finer granularity as whole source code files. In their evaluation, this approach outperform file based methods. Further, it reduces the number of source code lines to analyze in order to locate the defective parts.

A recent study of Akbar et al. investigated the history and development of IR-based tools for bug localization (Akbar and Kak 2020). The authors divided the tools in three major generations, (1st gen.) bag-of-words models, (2nd gen.) augmented bag-of-words models, and (3rd gen.) models based on exploitation of proximity and term to term relationships. The study concludes, that the third-generation of tools is superior to the former ones. The most advanced of them is SCOR (Akbar and Kak 2019). Here, the bag-of-words model is replaced with word embeddings to encode the terms of bug reports and source code files. Additionally, Markov Random Fields (MRF) are used to model term-term ordering constraints. The MRF framework is an undirected graph with one node representing a particular source code file and the other ones the terms of the query, i.e., the bug report. The edges among the nodes represent probabilistic dependencies between the nodes. This allows to model for example "full independence" (all query terms are independent), or "sequential dependence" by connecting subsequent query nodes. The evaluation of SCOR shows, that using MRF improves bug localization compared to approaches based on the bag of words model.

3.1.1. Datasets and Collections of Projects used to Evaluate Approaches

An overview of datasets and projects used to evaluate the reported approaches is provided in Table 3.1. There is only one dataset specifically designed for bug localization, iBugs (Dallmeier and Zimmermann 2007a), but it is barely used. Some approaches, i.e. BugScout, claim to use it, but report different numbers of bug

reports. However, different algorithm are often evaluated on similar projects. The other datasets are just enumerated for naming and represent the utilized projects and number of bug reports used in evaluation of an approach. Often projects stem from a common pool, such as AspectJ or Eclipse, but with a varying amount of bug reports and it is rather unclear how the selection was performed. An overview of utilized projects and number of bug reports is shown in Table 3.1. Beside iBugs, the dataset names are grouped into clusters and labeled **A** to **H**. This allows to compare different algorithms using the same selection of projects and bug reports (see Tab. 3.2), which will be discussed in the next section.

Table 3.1.: Summary of datasets and projects used to evaluate bug localization algorithms. There is only one dedicated dataset, iBugs (Dallmeier and Zimmermann 2007a, 2007b), the other ones are labeled, A to H, clusters of projects. These represent projects and the number of bug reports (in parentheses) used to evaluate a specific algorithm.

Name / Label	Projects and number of bug reports
iBugs	AspectJ (369)
A	ArgoUML (1,764), AspectJ (271, but claims it is **iBugs**), Eclipse (4,136), Jazz (6,264)
B	AspectJ (286, subset of **iBugs**), Eclipse (3,075), SWT (98), ZXing (20)
C	Firefox modules: ff-bookmark (1,927), ff-general (1,289) Mozilla-Core modules: core-js (2,391), core-dom (1,050), core-layout (2,391), core-style (1,131), core-xpcom (1,059), core-xul (1,260)
D	AspectJ (286), Eclipse (3,075), SWT (98)
E	AspectJ (593), Birt (4,178), Eclipse (6,495), JDT (6,274), SWT (4,151), Tomcat (1,056)
F	Birt (583), Eclipse (1,656), JDT (632), SWT (817)
G	AspectJ (244), JDT 4.5 (94), PDE 4.4 (60), SWT 3.1 (98), Tomcat 8.0 (193), ZXing (20)
H	29 projects including AspectJ (291) and Eclipse (4,035) with $\approx$ 20,000 bug reports in total

3.1.2. Comparison of IR-based Bug Localization Algorithms

An overview of published IR-based bug localization algorithms is given in Table 3.2. It also contains which dataset or project collection was used for evaluation (see Tab. 3.1), and the outcome of the evaluation, i.e. which algorithm performs best. Further, novel ideas of the respective algorithm are highlighted. The great variety of

algorithms and used datasets makes it difficult to judge and compare the different approaches. Without repeating all the studies, only statements about algorithms using the same dataset can be made. For example, out of the algorithms BugLocator, BLUiR, AmaLgam, and AmaLgam+, all evaluated on project collection labeled **B**, the latter performs best. But, this gives no clue when comparing AmaLgam+ and Locus, although both outperform AmaLgam in their respective evaluation. Thus, it is not clear which of the listed bug localization algorithms performs best overall.

Table 3.2.: Overview of different IR-based bug localization approaches. The first column assigns a name to the approach for identification. The second column states the used dataset (see Tab. 3.1). The column ”novelty” highlights novel techniques and designs differentiating the respective approach from competitors. The last column states the evaluated algorithms in the publication. The notation ”X > Y” summarizes the compared approaches and which one performed best: X in this case.

Approach	Dataset	Novelty	Evaluation outcome
BugScout (Nguyen et al. 2011)	**A**	LDA topic model.	BugScout > Lukins (LDA) (Lukins, Kraft, and Etzkorn 2008) > SVM
BugLocator (Zhou, Zhang, and Lo 2012)	**B**	Considering information of similar bug reports.	BugLocator > ”SUM approach”
BLUiR (Saha et al. 2013)	**B**	Structured IR on code constructs.	BLUiR > BugLocator
AmaLgam (Wang and Lo 2014)	**B**	Combining BugLocator and BLUiR.	AmaLgam > BLUiR > BugLocator
AmaLgam+ (Wang and Lo 2016)	**B**	Based on AmaLgam, but also considers stack traces, and bug reporter information.	AmaLgam+ > AmaLgam > BRTracer > BLUiR > BugLocator
Usual Suspects (D. Kim et al. 2013)	**C**	Baseline model suggesting most frequently modified source code files.	-
”Kim et al.” (D. Kim et al. 2013)	**C**	Two phase model, which only predicts source code files when bug report contains sufficient information.	”Kim et al.” > BugScout > Usual Suspects
BRTracer (C. Wong et al. 2014)	**D**	Analysing stack traces found in bug reports.	BRTracer > BugLocator

Approach	Dataset	Novelty	Evaluation outcome
Learning to rank (LR) (Ye, Bunescu, and Liu 2014)	**E**	Using learning-to-rank technique.	LR > BugLocator
HyLoc (Lam et al. 2015)	**E**	Using deep neural networks.	HyLoc > LR > BugLocator
LR+WE (Ye et al. 2016)	**F**	Using learning-to-rank technique and word embeddings.	LR+WE > LR > ”Kim et al.”
Locus (Wen, Wu, and Cheung 2016)	**G**	Sub source code file bug location granularity, and usage of code change history.	Locus > AmaLgam > BLUiR > BR-Tracer
SCOR (Akbar and Kak 2019, 2020)	**H**	Combining word embeddings with Markov Random Fields.	SCOR > BLUiR > BugLocator

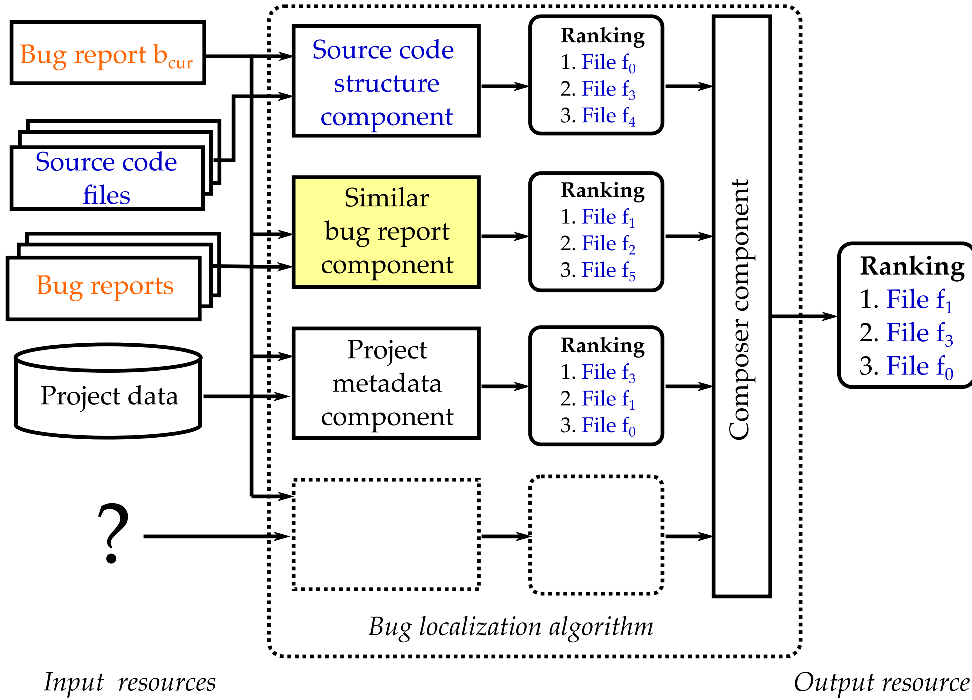


Figure 3.1.: Identified structure of a modern bug localization framework.

3.1.3. Internal Structure of Bug Localization Algorithms

After studying the state-of-the-art a framework for bug localization algorithms was derived as shown in Figure 3.1. It consist of different components and each component analyzes a specific input resource. The resources are internally processed by the dedicated components and result in multiple, intermediate source code file rankings, reflecting the bug localization only based on the respective input resources. Finally, all individual rankings are aggregated by a **composer component** generating the final source code file ranking. Over the years, different aggregation methods to combine the rankings have been proposed, including summation (Saha et al. 2013), empirically determined weighting schemes (Zhou, Zhang, and Lo 2012; Wang and Lo 2014), weights trained by vector support machines (Ye, Bunescu, and Liu 2014), and neural networks (Lam et al. 2015).

The most common component is the **source code (structure) component**, which directly relates a bug report to the contents of the projects source code files. Thus, the bug report and each source code file are treated as text files and the component calculates a similarity score for each file using IR techniques. Various text matching algorithms have been proposed for this purpose (Dit et al. 2013). The resulting similarity score determines the ranking in the output list. More sophisticated methods

preprocess the source code and distinguish individual source code parts, such as identifiers, class and method names, and comments (Saha et al. 2013). Next, these representations of the source code file are used for similarity analysis to the bug report.

Studies showed that bugs often occur in bursts and not in isolation (S. Kim et al. 2007). Thus, the **version history component** tries to expose this observation. The key idea is, that source code files responsible for a bug recently are more likely to be responsible for new bugs in the (near) future.

Next to bug reports and source code files, researches also utilized metadata resulting in different project **metadata component**s. For example, the bug reports' project version, platform and priority is used (D. Kim et al. 2013; Ye, Bunescu, and Liu 2014), or the recency and frequency the source code files were modified as part of bug fixing activities (Lam et al. 2015). Other often leveraged information is additional textual input, i.e. API documentation (Lam et al. 2015; McMillan et al. 2012; Ye, Bunescu, and Liu 2014; Ye et al. 2016).

The **similar bug report component** takes previously resolved bug reports as input next to the current one. The assumption is, that the same set of source code files are potential candidates for modifications in case the bug report is similar to an already resolved one.

3.2. Localizing Features and Recovering Trace Links

Manually creating and maintaining trace links is associated with high costs (Heindl and Biffl 2005), and researchers studied the application of IR-based methods to support automated trace recovery (Cleland-Huang et al. 2007; De Lucia, Fasano, and Oliveto 2008; Keenan et al. 2012; Mahmoud and Niu 2010, 2011; Niu et al. 2014; Rempel, Mäder, and Kuschke 2013). Hayes et al. apply a vector space model (VSM) in combination with a thesaurus to establish new trace links between requirements (Hayes, Dekhtyar, and Sundaram 2006). Further, approaches based on Latent Semantic Indexing (LSI) (A. D. Lucia et al. 2004; Rempel, Mäder, and Kuschke 2013) and Latent Dirichlet Allocation (LDA) (Asuncion, Asuncion, and Taylor 2010; Dekhtyar et al. 2007) were developed, as well. Guo et al. used a recurrent neural network to integrate semantics or context in which various terms are used (Guo, Cheng, and Cleland-Huang 2017). Instead of relying on a single technique, researchers also combined the results of individual algorithms (Dekhtyar et al. 2007; Gethers et al. 2011; Lohar et al. 2013), and used AI swarm techniques to retrieve trace links (Sultanov, Hayes, and Kong 2011). Any automated trace recovery approach implies the risk to create incorrect trace links (Merten et al. 2016; Knethen et al. 2002).

Thus researchers leveraged structural artifact Information to improve the correctness of the recovered trace links and therefore tackle the traceability quality problem (Panichella et al. 2013).

Another related area is feature localization, which attempts to identify sections of source code related to a specific requirement or issue. A probabilistic model and a VSM approach to retrieve trace links between code and documentation was proposed in (Antoniol et al. 2002). Poshyvanyk et al. (Poshyvanyk et al. 2006) treated the feature localization as a decision-making problem in the presence of uncertainty. They combined a probabilistic model and LSI to identify traces from features, i.e. functionality of a system usually captured by requirements, to parts of a project's source code. The systematic literature review of Dit et al. (Dit et al. 2013) provides a comprehensive overview of feature location algorithms. A large body of work covers IR-based techniques, including VSM, LSI, and LDA. The authors summarize, the quality of feature localization is heavily tied to the quality of the source code naming conventions and the users-issued query, i.e. the lexical gap. Researchers also investigated the use of dynamic analysis for feature localization (Kuang et al. 2015; Kuang et al. 2012; Kuang et al. 2017). The combination of static and dynamic analysis is studied in (Eisenbarth, Koschke, and Simon 2003) to rapidly focus on the system's parts that relate to specific features.

Instead of directly tracing from issues (requirements or bug reports) to source code files researchers also explored ideas to trace to commits as an intermediate step. Le et al. proposes *RCLinker* which utilizes textual features and metadata features to establish the trace links (Le et al. 2015). The textual features are based on *ChangeScribe* which automatically generates commit messages by analyzing the performed source code changes between two commits. The metadata features are derived from issue such as its priority or the number of attached comments. All extracted features are used to train a random forest classifier, which is able to predict whether an issue and a commit should be linked. The approach is evaluated on six Java projects from the apache software foundation. *FRLink* revisits RCLinker by also including non-source code documents and excluding irrelevant source code files to reduce data noise (Sun, Wang, and Yang 2017). The evaluation shows that FRLink outperforms RCLinker in term of F-measure. Ruan et al. propose *DeepLink* which outperforms FRLink (Ruan et al. 2019). Deeplink utilizes a semantically enhanced issue-to-commit trace link recovery method based on recurrent neural networks. Therefore traditional approaches to represent texts like VSM are replaced by word embeddings, which then are used as inputs for the network.

3.3. Criticizing the State of the Art

The previous sections outlined how IR-based bug localization algorithms and trace link recovery approaches evolved over the past years and became more and more sophisticated. But, one can argue that there is still room for improvement. First, most approaches focus on improving performance soley based on textual similarity between bug reports and source code files. Second, the bug localization algorithms only utilize bug report artifacts at a fixed point in the projects' development history. Internally, this is handled by the similar bug report component of the algorithms (see Sec. 3.1.3). However, this component has not been improved during the last years and was not subject for detailed research compared the other components (including the composer component).

Agile software development is driven by constantly implementing small enhancements and thus altering the project's code base. These past activities probably also introduce software bugs, but this knowledge is not leveraged. Further, knowing the code places realizing specific requirements has high potential to be benefitial for bug localization. The assumption is that bug reports refer to requirements, either explicitly via trace links or implicitly by textual similarity, and thus provide an initial guess where to look for bugs in the source code files. Thus the similar bug report component should be improved to a similar *issue* component.

Last, traceability information is not utilized. Especially, the time consuming tagging procedure of commit messages creates valueable information about issue artifacts and source code. As described, there are several algorithms to establish trace links from issue artifacs to commits, but they mainly focus on textual similarity. Knowledge about the project's workflow, lifecycle, and stakeholder interaction is not leveraged. Futher, the approaches end after establishing trace links without further using them, and thus can be seen as building blocks for advanced algorithms.

To sum up, there is no holistic approach utilizing a project's historic issue artifacts combined with augmented trace links among them to tackle the downstream task of bug localization.

4. A Holistic Approach to Improve IR-based Bug Localization

This section outlines the overall idea of this thesis consisting of three stages. The following subsections outline the stages and a formal notation and artifact model used throughout the rest of this thesis is introduced.

4.1. Outlining the Idea

Leveraging requirements, trace links among issue artifacts, and considering the project's history are the main assumptions to improve the challenging task of bug localization within this thesis. Creating trace links among development artifacts, especially among issues-to-commits, is a common practice in todays development process. However, the necessary steps to establish these trace links are performed manually and thus subject for failures.

Figure 4.1 provides an overview of the holistic approach to improve IR-based bug localization. The projects' ITS and VCS are used to derive a unified artifact model which is described in Section 4.2. This model and the input sources are *mined* to create the *SEOSS*[1] dataset, which is used throughout this thesis. The mining process and details of the dataset are described in Section 5. The SEOSS dataset was specifically created to study the algorithms created in thesis, because no suitable datasets were available. Section 6 uses the artifact model to derive a traceability graph model. A novel similar issue component *TraceScore* is derived by analyzing the graph. Further, a novel source code structure component, *LuceneScore*, is also proposed in this section. Both components are utilized in the bug localization algorithm *ABLoTS*. Section 7 defines a *tagging model* as foundation for the *trace link set augmentation (TSLA)* algorithm. This algorithm is used to augment the traceability graph, i.e. add missing issue-to-commit trace links. Finally, all proposed algorithms are evaluated in Section 8.

1. Software Engineering in Open-Source Systems

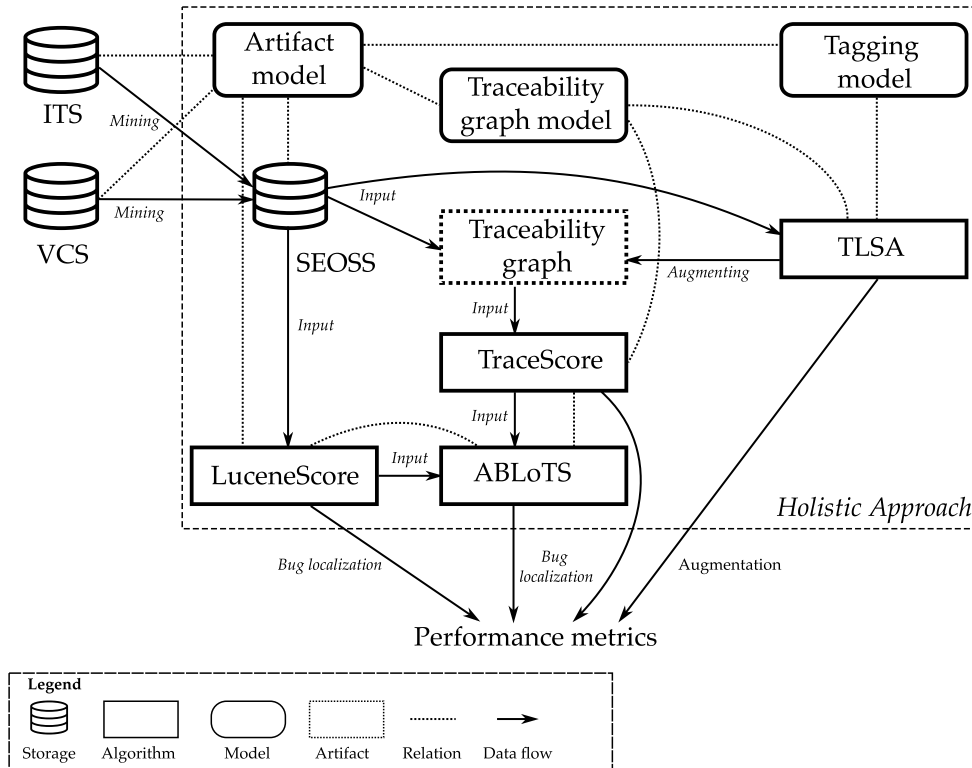


Figure 4.1.: Overview of the holistic approach to improve IR-based bug localization.

4.1.1. Creating a Dataset to Evaluate Bug Localization Algorithms

Over the years, the development of IR-based localization algorithms led to a couple of datasets and projects used for evaluation and benchmarking. Tables 3.1, 3.2 show that there currently is no large benchmark dataset consistently used to compare bug localization algorithms. The iBugs dataset, specially designed for bug localization tasks, contains just one project with only 369 bug reports (Dallmeier and Zimmermann 2007a). Further, none of the mentioned datasets does contain any trace link information, or project history. They consists of a set of curated bug reports, a snapshot of the projects source code files, and the *oracle* (truth) information, which source code files were modified to fix the respective bug. Additionally, their size is often limited and thus less suitable for machine learning algorithms, which require large amounts of data for training and testing. Therefore there is the need to create a new dataset, which is described in Section 5.

4.1.2. Designing a Bug Localization Algorithm utilizing Traceability

Studying the state of the art of bug localization algorithms revealed (see Sec. 3.1), that no approach exists which leverages projects trace links, especially those found in ITS. Section 6 describes the design and implementation of the new approach to incorporate traceability information to bug localization. The newly created dataset will be used to evaluate the approach.

4.1.3. Augmenting the Issue-to-Commit Trace Link Set

Creating and maintaining trace links requires a lot of effort and time. Thus, the amount of available trace links found in projects is limited. In open-source projects it is a good practice to tag commit messages with issue identifiers to establish trace links, but often traceability is not mandated or even prescribed. This may additionally result in less or low quality trace links. Section 7 explores the idea to augment the existing set of issue-to-commit trace links in a project. The result is enriched project data which may be beneficial for the developed bug localization algorithm.

4.2. Constructing an Artifact Model

Issue Tracking Systems and Version Control Systems contain a variety of artifacts as introduced in Sections 2.4.1, 2.4.2. After studying the concepts of Jira and git, the following artifact model was constructed and is used in this thesis (see Fig. 4.2).

4.2.1. Describing Contained Artifacts

The model consists of issues from the ITS, and commits and source code files from the VCS. The set of issues of a projects is denoted as $\mathcal{I}$. Out of them, the subset $\mathcal{B} \subseteq \mathcal{I}$ denotes all bug reports, and the set $\mathcal{R} = \mathcal{I} \setminus \mathcal{B}$ represents all requirements. The set of commits $\mathcal{C}$ stem from the projects' VCS. Different kinds of files may be contained within each commit such as source code modifications, changed documentation, added examples, and tests. However, out of all project files $\mathcal{F}$, only the subset of source code files is relevant within this model.

Lowercase letters are used to identify individual artifacts, such as the current bug report at hand $b_{\text{cur}} \in \mathcal{B}$, or a source code file $f \in \mathcal{F}$. Uppercase letters are used for subsets, such as $I \subset \mathcal{I}$.

4.2.2. Describing Artifact Relations

Issue tracking systems allow to create typed links between issues. These trace links are *explicitly* created and maintained by the project's developers. For example, it is used to model dependencies between issues (one feature needs to be implemented before another one), or simple relations (a bug report relates to a feature). Since traceability is often not mandated, especially in open-source systems, this set of trace links may not be complete, i.e. there are issues not tracing to any other artifacts.

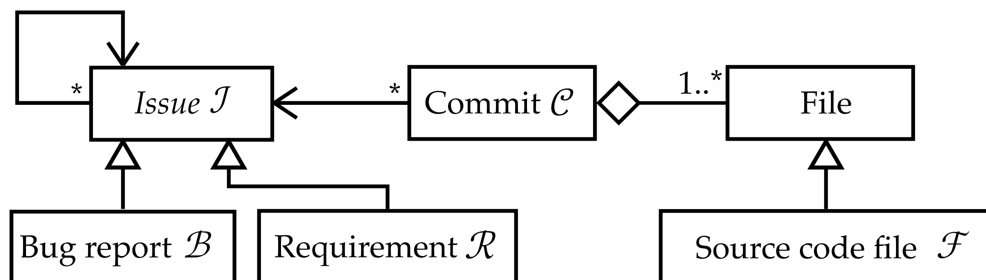


Figure 4.2.: UML diagram of used artifact model.

Bundling file modifications in commits is an inherent feature of VCS. After changing a set of source code files, a developer commits this changeset to the VCS. In this process, an atomic commit is created and enriched with additional metadata, i.e. a timestamp, a commit message and information about the developer performing the changes.

There is a relation between issues and commits, which is established using the tagging procedure. To associate a commit to an issue, the developers place the unique issue identifier within the commit message.

The model enables to trace from issues to individual source code files. This allows to identify all source code files that have been modified to fix a bug, or where a specific requirements is implemented in the source code. Several functions and sets are used to express the outlined process and applied methodologies in a formal way. Assuming $i \in \mathcal{I}, c \in \mathcal{C}, f \in \mathcal{F}$, the following functions are defined.

$$\begin{aligned}
\text{isLinked} &: \mathcal{I} \times \mathcal{C} \rightarrow \{0, 1\} \\
\text{isLinked}_\text{I} &: \mathcal{I} \times \mathcal{I} \rightarrow \{0, 1\} \\
\text{commits}(i) &= \{c | \,\text{isLinked}(i, c) = 1\} \\
\text{changed}_\text{C}(c) &= \{f | c \text{ contains } f\} \\
\text{changed}_\text{I}(i) &= \{\text{changed}_\text{C}(c) | \,\text{isLinked}(i, c) = 1\} \\
\text{created} &: i \rightarrow \text{t} \\
\text{resolved} &: i \rightarrow \text{t} \\
\text{committed} &: c \rightarrow \text{t}
\end{aligned} \tag{4.1}$$

The function isLinked returns 1, if a given issue and commit are linked, i.e., a developer tagged the respective commit message with the issue identifier. Similarly, isLinked_I is 1 if a trace link in the ITS exists between two issue artifacts. The function commits returns all linked commits for a given issue. The functions changed_C and changed_I return all source code files that belong to a given commit, or that were changed in order to resolve a given issue. Finally, the time-related functions created and resolved return the respective values when an issue was created or resolved, and committed returns the time a commit occurred.

4.3. Summary

This section outlined the three steps of the holistic approach to be developed in this thesis. Further, a formal definition of the underlying artifact model was given.

5. Mining Software Repositories to Create Holistic Datasets

Related Publications This section is based on the following publications

> **Paper I** (Rath, Rempel, and Mäder 2017): M. Rath, P. Rempel, and P. Mäder "The IlmSeven Dataset" in RE, pages 516-519, 2017
>
> **Paper IV** (Rath and Mäder 2019b): M. Rath and P. Mäder "The SEOSS 33 dataset - requirements, bug reports, code history, and trace links for entire projects" in Data in Brief, Vol. 25, 2019

This section describes mining of open-source projects. The lack of suitable datasets for the novel bug localization approach required to create a new dataset: *Software Engineering in Open-Source Systems (SEOSS)*. It contains different software artifacts from issue tracking and version control systems of 33 projects and captures the artifacts' history. All artifacts are stored in a database on a per project basis. The uniform schema of these databases provides a convenient way to access and study the stored data.

Large amounts of data are required to perform solid scientific investigations. In this thesis, the data is artifacts resulting from activities performed during the software and systems development. The artifacts include feature requests, bug reports, commits, and source code files. Collecting the data initially requires access to representative data sources. One potential data source are commercial and industrial projects. However, due to multiple concerns including intellectual property, and competition, commercial project owners are not willing to give researchers access to their data (Hassan 2008). With the rise of open-source software during the last decade, large amounts of projects are freely available. This provides unrestricted access to issue tracking systems and source code repositories of projects with manifold characteristics (Kaur and Vig 2016). Next to the source code stored in VCSs, the projects use ITS to manage requirements artifacts and bug reports. All this information is stored in heterogeneous systems in different formats and thus is difficult to directly process the data. The process of extracting information about the artifacts and their relationships produced and archived during the evolution of a software system is called *mining software repositories (MSR)* (Kagdi, Collard, and Maletic 2007). However, most repositories have no direct support for mining activities. Implementing an automatic

data extraction for software repositories is complex, takes a lot of effort and time and are major obstacles for researchers. Actually, many researchers report, that the mining task is a barrier in the software engineering domain in general (Liebchen and Shepperd 2016). This task becomes even more difficult in case a project uses multiple connected repositories (probably dissected by specific software components), like distributed ITS and separate VCS. Therefore, researchers request convenient access to already mined repositories available in an easy way to process (Hassan 2008).

The mining framework developed for this thesis evolved over time and was constantly improved. It first was used to create and publish the *IlmSeven Dataset* (Rath, Rempel, and Mäder 2017). Afterwards, multiple studies about traceability also benefited from its capabilities (Rath, Goman, and Mäder 2017; Tomova, Rath, and Mäder 2017; Goman, Rath, and Mäder 2017; Tomova, Rath, and Mäder 2018). Initially only a project's ITS could be mined. However, also mining the project's VCS was added to permit studies that span a wide range of the software development process. Eventually this led to the creation of the SEOSS Dataset (Rath and Mäder 2019b).

The reminder of this section is structured as follows. First, the project selection method is described. Afterwards, the implemented mining process is shown as well as the storage format of the created SEOSS dataset. The section ends with a formal definition of the dataset.

5.1. Selecting Projects for Mining

Different ITS and VCS implementations exist. Thus, the selection of projects for mining also includes which ITS and VCS should be supported by the mining process. Supporting many ITS and VCS combinations increases the complexity of the mining procedure. Therefore it was decided to, at first, only support one ITS and VCS, which simplifies the development of the data collection process. However, this (initial) limitation could later be resolved by adapting the mining tool chain to other systems, which mainly requires time and effort. Based on popularity (Project Management Zone 2018; Stackoverflow 2018), ITS Atlassian Jira and VCS Git were chosen.

Accounting for the ITS selection, three major hosting providers were considered to retrieve projects: the Apache Software Foundation, JBoss, and the Atlassian Cloud. A pre-study was conducted and provided a preliminary, quantitative overview of the artifacts and trace links in hosted projects. Finally, the following selection criteria were defined:

1. All projects should mainly be written in the same programming language. This simplifies later analysis once the dataset is created. After studying programming language polls (GitHub, Inc. 2018; IEEE Spectrum 2017), and Internet searches (TIOBE Software BV 2021), the Java programming language was selected.
2. Each project shall contain a rich set of artifacts, and the bug report, requirement and source code file artifacts should be present.
3. The project shall continuously capture trace links within the ITS, i.e. among the issues, as well as from ITS to VCS.
4. A project shall have deployed stable releases, and shall be under active development for at least three years.

Using these criteria, 33 projects (Rath and Mäder 2019b) were chosen by applying the information oriented selection strategy Maximum Variation Cases (Flyvbjerg 2006). This methods draws representative samples for project's characteristics, such as the amount of artifacts, number of existing trace links, and issue types. The so created dataset is available online[1].

5.2. Collecting Project Artifacts from Multiple Repositories

The collection process previously designed for the *IlmSeven Dataset* (Rath, Rempel, and Mäder 2017), which targeted projects to primarily study traceability was refined to create SEOSS. The result is a sequential capturing process to gather data per project as shown in Figure 5.1).

5.2.1. Analyzing a Project's Issue Tracking System

The first collector is used to extract data from the ITS. It is targeted to operate with the Jira platform, which offers a RESTful web service. This allows to interact with the platform programmatically using the provided application programming interface (API). The collector automatically discovers all project's artifacts and stores them locally in *JavaScript Object Notation (JSON)* format. The downloaded files are analyzed and contained artifact data and trace links are extracted. Lastly, the collected data is stored in respective tables (see Sec. 5.3) of the project's database.

Jira ITS uses a common prefix, typically derived from the project's name, for all artifacts inside the repository, e.g. "ZOOKEEPER" for project Zookeeper. The

1. https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/PDDZ4Q

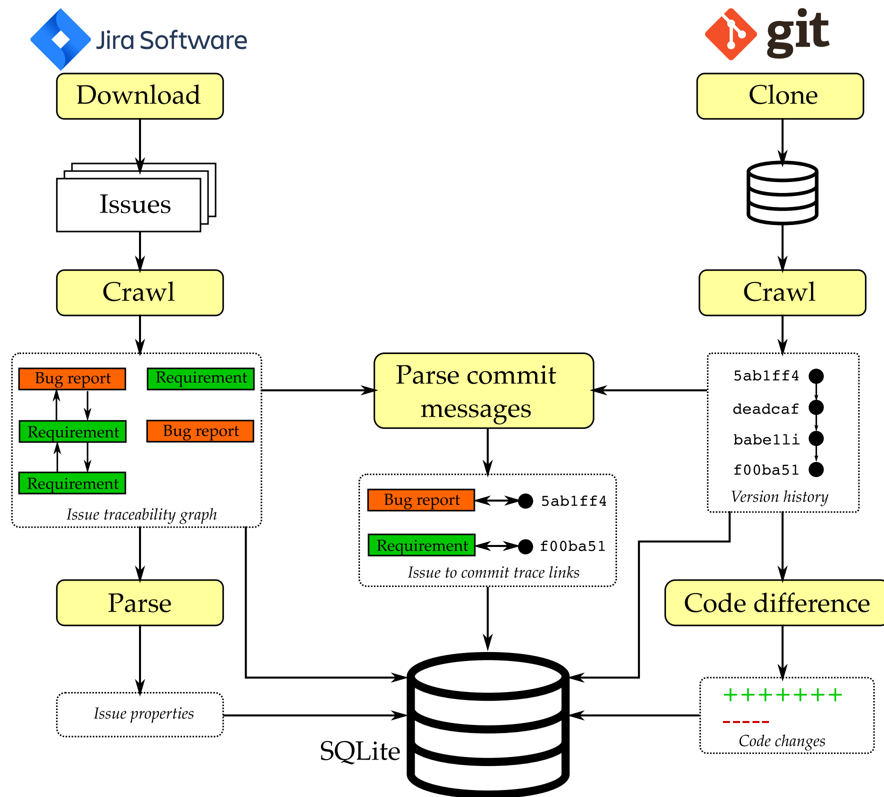


Figure 5.1.: The mining process applied per project in the dataset. It gathers information from each projects' ITS and VCS, processes it, and finally stores it in one database.

prefix is followed by an increasing number[2] and is used as a unique identifier for the artifact, the *issue id*. Usually, all project artifacts are located in a single repository, and thus share the same prefix. This is true e.g. for the projects Groovy, Maven, and Kafka in the SEOSS dataset. However, large projects utilize multiple repositories such as Hadoop. It consists of four repositories[3] with the prefixes HADOOP, HDFS, MAPREDUCE, and YARN, which all need to be mined. The repositories were determined by visiting the project's webpages, and the collector was applied to each of them. The references of utilized repositories are stored as meta data (see table meta in Fig. 5.2).

5.2.2. Analyzing a Project's Version Control System

A second collector gathers information about the commits from the project's VSC. The retrieved project commit history is traversed in reverse chronological order, starting with the most recent change, denoted as `HEAD` version of the master branch, back to the first recorded commit (initial version). For each visited commit during the traversal the commit properties are captured, i.e. the unique commit hash, the author, timestamp, and the names of the modified source code files. The content of the files is intentionally not stored. Depending on the project's size and duration of its development huge amounts of file changes may arise resulting in huge database files. But, Git is a content-addressable filesystem[4] and implements efficient mechanisms to minimize the required storage. For example, only file modifications (deltas) are stored, and not the whole file. The deltas are also regularly compressed to further reduce storage space. Thus, the collector only captures the number of added and deleted lines for each file change. This provides a first insight for the magnitude of the change. In case the actual content is required, the unique commit hash is used to query the Git. For example, the command `git show -pretty="" -U <commit-hash> <file-path>` shows the content for a given file and commit hash. Each commit message is scanned for unique issue identifiers to retrieve existing trace links to issues from the ITS (see Sec. 2.4.3). However, since tagging the commit messages is a manual process it is error-prone, and thus not all commits are tagged. This situation is extensively studied and discussed in Sec. 7. Eventually, the collected commit properties and identified trace links are stored in the database.

2. Regular expression pattern for Jira identifiers: https://bit.ly/2L2iZHg
3. https://hadoop.apache.org/issue_tracking.html
4. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

5.3. Organizing Project Artifacts in Unified Storage

Every project of the SEOSS dataset is individually stored in a relational database. SQLite[5] is used for this purpose. SQLite does not follow a client-server architecture and thus can be easily embedded into programs, which allows easy setup. Further, bindings for many programming languages exists simplifying working with the data. SQLite also claims to be the most widely deployed and used database engine in the world[6]. Figure 5.2 shows the database schema consisting of eight tables storing the collected artifact data and one meta table. A detailed description of these tables is shown in Table 5.1.

Table 5.1.: Description of all tables in a project's database. The relations among the tables is shown in Figure 5.2

Table	Description
`issue`	This table contains all artifacts extracted from project's issue tracker. Each issue includes a unique id, summary, a detailed description, information about type, status, involved developer names, and time stamps. The names are available as full names (e.g. "John Doe") and as login names (e.g. "jdoe"). These two values allow to relate users of the ITS to those in the VCS and identify same identities.
`issue_link`	This table contains directed, typed trace links between ITS artifacts. The type is stored in the `name` column (e.g. "Reference", or "Duplicate"), and the reading direction from the source to the target is stored in the `outward_label` column. The column `is_containment` captures, if the link represents a parent-child relationship.
`issue_comment`	This tables stores the comment attached to an issue. The commenting developer is identified by its full name and login name.
`issue_component`	A mapping table from issue to project specific software components (e.g. "web interface", "parser").
`issue_fix_version`	A mapping table containing the software version when the respective artifact was (or will be) fixed.
`change_set`	This table contains all commits mined from the project's VCS. The commits are uniquely identified by their `commit_hash`. Additional information such as the author of the commit (with full name and login name) and a textual description of the change are stored.

5. https://www.sqlite3.org
6. https://www.sqlite.org/mostdeployed.html

change_set_link	This table holds the parsed trace links between ITS and VCS artifacts.
code_change	This table contains the files modified for the commits. It stores the file name and the number added and removed lines. The actual content of the file is not part of the database. Files are identified by the file_path. In case the path was modified in the commit, the former path is available in old_file_path.
meta	A meta table containing information about the collection process. This includes the URLs of the processed ITS and VCS, time stamps when the mining occurred, and used Git HEAD commit hashes.

The database stores the collected information *as is*. For example, ITS texts may contain markup symbols[7] as defined by Jira. A few post processing steps are performed which only add information and do not alter the original data, and primarily simplify working with the dataset. For example, the time information may contain time zones (captured in the *_zoned columns). The time stamps are also available in universal coordinated time (UTC) for simpler comparison. Further, ISO 8601 encoding[8] is used to represent time values (e.g. "2021-05-04T11:22:33"), instead of *locale* specific formatting (e.g. "11:22:33am 05/04/21").

5.4. Key Figures of the Created Dataset

Table 5.2 depicts key figures for each project in the dataset. Each row represents a project, and the columns depict the number of contained issues, the number of commits, the amount of issue-to-issue trace links, and finally the number of trace links from issues to commits. Overall it contains about 280.000 issues, 360.000 commits, 100.000 issue-to-issue links, and 200.000 issue-to-commit links. Based on the selection criteria, the projects represent various combinations for the measures. For example, there are large projects in terms of issues, such as Hadoop, Hbase, or Spark. Project Cassandra has nearly the same amount of commits as Hadoop, but only a third of the issues. There are projects with few links (issue-to-issue, issue-to-commit), like in project Errai, and ones with multiple thousands as in project Wildfly.

7. https://bit.ly/2Fh8Vd8
8. https://www.iso.org/iso-8601-date-and-time-format.html

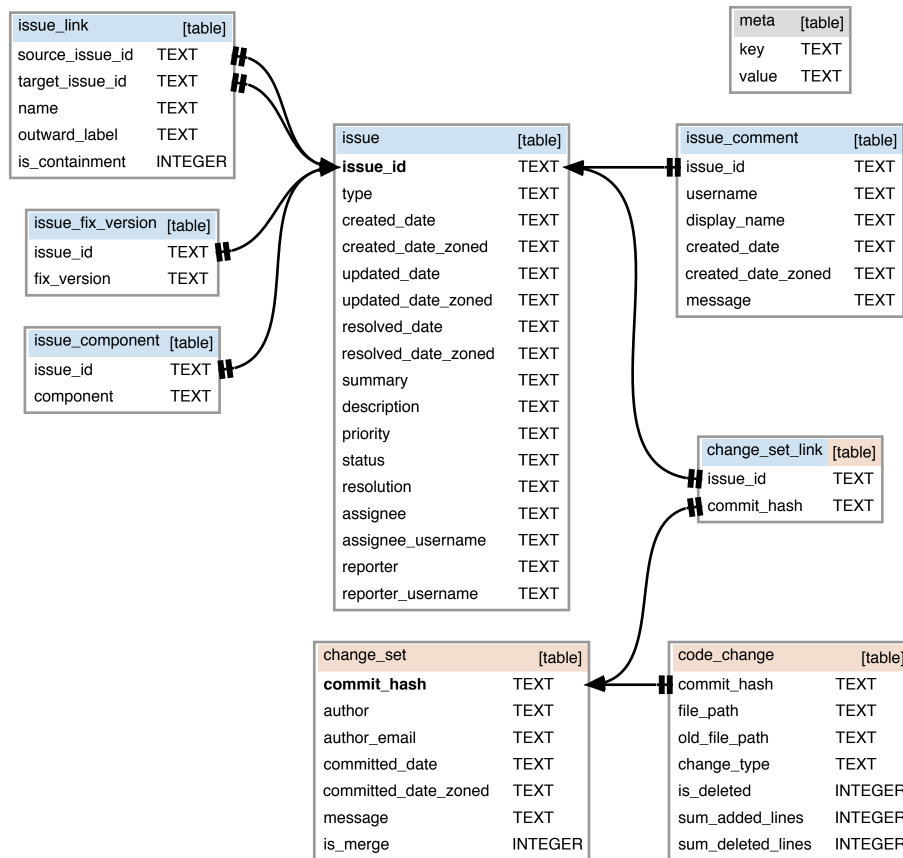


Figure 5.2.: Database schema used for each project in the dataset. Primary keys are bold, and arrows depict foreign key references among the tables. The tables highlighted in ■ contain data retrieved from the project's ITS, and those highlighted in ■ from the project's VCS. The table `change_set_link` holds the identified trace links which connect the artifacts from the two systems.

Table 5.2.: Key figures for the projects of the SEOSS dataset.

Project	#Issues	#Commits	#Issue-to-issue trace links	#Issue-to-commit trace links
Archiva	1,929	8,006	510	2,275
Axis2	5,796	13,114	484	2,609
Cassandra	13,965	23,592	3,363	8,835
Derby	6,969	8,156	3,529	7,263
Drools	5,103	11,117	633	5,528
Errai	1,060	7,645	40	638
Flink	8,100	12,419	1,895	5,328
Groovy	8,137	12,378	987	4,901
Hadoop	39,086	27,776	23,626	29,309
Hbase	19,247	14,331	7,694	13,379
Hibernate	11,971	8,173	3,110	6,942
Hive	18,025	11,179	9,110	11,058
Hornetq	3,286	11,121	732	2,283
Infinispan	8,422	10,654	2,350	7,499
Izpack	1,337	5,489	73	1,804
Jbehave	1,243	3,282	4	1,546
JTM	2,887	2,204	814	1,615
Jbpm	10,397	4,919	2,211	1,133
Kafka	6,219	4,426	1,699	2,893
Keycloak	5,523	10,106	1,408	5,443
Log4j2	2,114	9,792	488	3,309
Lucene	17,329	28,995	4,923	19,111
Maven	5,073	10,315	1,971	2,596
Pig	5,234	3,134	1,479	2,980
Railo	3,326	3,990	45	989
Resteasy	1,649	3,684	280	1,345
Seam2	5,031	11,309	953	3,024
Spark	22,205	20,829	8,070	14,306
Switchyard	3,010	2,928	882	2,428
Teiid	4,899	7,266	723	5,858
Weld	2,518	8,086	665	2,555
Wildfly	24,566	36,711	11,033	22,388
Zookeeper	2,907	1,600	966	1,423
$\sum$	278,563	358,726	96,750	204,593

5.5. Defining the Mined Dataset

A formal definition for the projects $\text{Proj}_p^{(\text{seoss})}$, with $p \in \mathsf{SEOSS}_{\text{project-names}} = \{\text{Archiva}, \dots, \text{ZooKeeper}\}$ in the SEOSS dataset is

$$
\begin{aligned}
p &\in \mathsf{SEOSS}_{\text{project-names}} \\
\text{Proj}_p^{(\text{seoss})} &= \left(I_p^{(\text{seoss})}, C_p^{(\text{seoss})}, F_p^{(\text{seoss})}\right) \\
\mathsf{SEOSS} &= \bigcup_p \text{Proj}_p^{(\text{seoss})}
\end{aligned}
\tag{5.1}
$$

A $\text{Proj}_p^{(\text{seoss})}$ is a tuple containing issues I, commits C, and source code files F as described in the artifact model (see Sec. 4.2). SEOSS represents the SEOSS dataset containing 33 project tuples.

5.6. Summary

Conducting empirical studies requires (large) datasets. The lack of datasets containing a reasonable amount of requirements, commits, source code files and the trace links among them motivated to create the SEOSS dataset. It contains the mentioned artifacts of 33 open-source projects. The artifacts were collected by mining the projects' issue tracking systems and version control systems. This section first described how the 33 projects were selected to get a representative sample out of existing projects. Next the design and implementation of the fully automated mining process was explained. It consists of two individual data processing pipelines handling the issue tracker artifacts and version control system artifacts respectively. An interlinking processing step discovers trace links among the artifacts of the two pipelines. All collected data is stored in a per project database. Afterwards, the database schema, which is identical for all projects, was discussed. Next, key figures and characteristics of the mined projects were presented. The section ended with a formal definition and notation of the SEOSS dataset, which will be used in the remainder of this thesis. The SEOSS dataset serves as an essential foundation to develop, investigate, and study the presented algorithms, especially the novel bug localization algorithm, of this thesis.

6. The ABLoTS Bug Localization Approach

Related Publication This section is based on the publication

> **Paper II** (Rath, Lo, and Mäder 2018): M. Rath, D. Lo, and P. Mäder. "Analyzing requirements and traceability information to improve bug localization." in Mining Software Repositories (MSR), pages 442–453, 2018

A novel component for an IR-based bug localization algorithm is developed in this section. It utilizes bug reports, issue artifacts and existing issue-to-issue trace links mined from a project's ITS. With this information, a *TraceScore* is calculated for source code files leveraging the bug report at hand, previously resolved bug reports and requirements, and the trace links among these artifacts. The resulting source code file rankings created by this score outperforms existing approaches which also incorporate previously fixed bug reports (see Section 8.4). Plugged into the *Automated bug localization algorithm using TraceScore (ABLoTS)*, TraceScore boosts the overall IR-based bug localization performance.

The hypothesis of this section is, that the "similar report" component of a bug localization algorithm (see Fig. 3.1) could be considerably improved, by leveraging two additional information resources. **First**, instead on only focusing on previously filed bug reports, the whole history of the development process and created artifacts should be leveraged. Especially the issues from the ITS are interesting in this regard. Using all issues overcomes the problem, that the similar report component is only capable to propose source code files, *if* these were already part of a previous bug fix. Instead, source code files modified by other software changes, such as implementing a new feature, introduce bugs as well (Sliwerski, Zimmermann, and Zeller 2005; Kim, Jr., and Zhang 2008). Because of this scope extension beyond bug reports, the novel component is classified as similar *issue* component throughout this section. **Second**, the component should utilize the established trace links between artifacts. For example in modern system development, once a requirement or improvement is implemented, trace links to the changed source code files are created (Schermann et al. 2015; Rath, Rempel, and Mäder 2017). This also applies to bug reports such that the modified source code files for previously fixed bugs are identifiable. Thus, related artifacts can be detected *independent* from their textual representation utilizing trace links. However, as complete traceability is difficult to create and maintain, textual similarity can be used as fall-back approach.

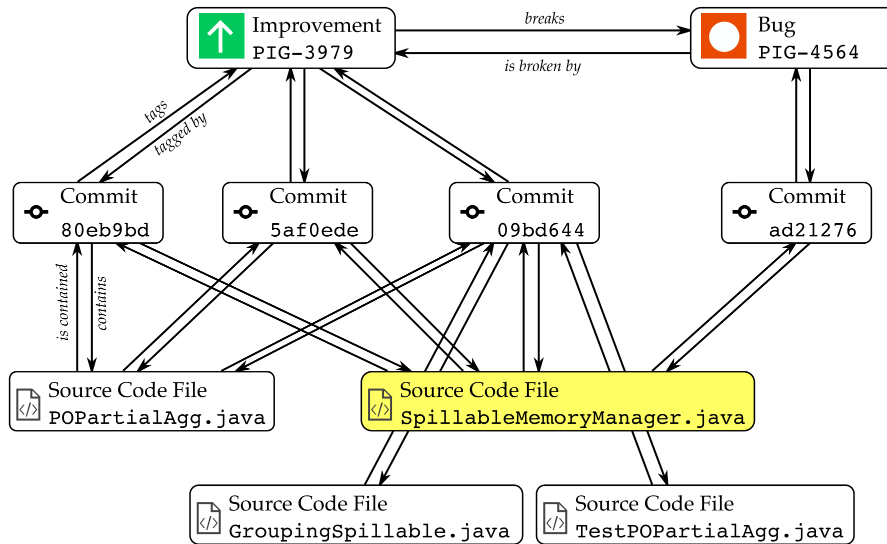


Figure 6.1.: Traceability graph example in project Pig. The implementation of improvement PIG-3979 modifies source code file SpillableMemoryManager.java and introduces a bug. This is later removed when resolving bug report PIG-4564, which is explicitly traced to improvement PIG-3979. Thus considering source code file changes introduced by implementing requirements may provide valuable information for bug localization. The novel algorithm TraceScore introduced in this section exposes this observation.

6.1. Motivating Example to Leverage ITS Project Data for Bug Localization

Figure 6.1 shows the traceability graph for issue artifacts PIG-3979[1] and PIG-4564[2]. The requirement (typed as "improvement") PIG-3979 modifies four source code files, distributed over three commits, and is *explicitly* traced to bug report PIG-4564. Implementing the improvement introduced a bug in source code file SpillableMemoryManager.java, which is then fixed by resolving bug report PIG-4564. Thus also considering requirement documents in the bug localization process provides valuable information to localize bugs in the projects' source code.

1. https://issues.apache.org/jira/browse/PIG-3979
2. https://issues.apache.org/jira/browse/PIG-4564

6.2. Multi-Component IR-Based Bug Localization Algorithms

Bug localization algorithms are usually composed of multiple components (see Sec. 3.1.3). This section analyzes the internal structure of two existing algorithms. Further, similar bug report components are studied. The results, especially the identified shortcomings, if these algorithms lead to the novel approaches presented in sections after this one.

6.2.1. Dissecting Existing Algorithms

This section takes a closer look at two bug localization algorithms. Especially the internal components, mechanics, and their interplay are reviewed. The algorithms were chosen, because they represent the most evolved IR-based approaches with published source code[3]. This easily enables their application on new datasets without relying on previously published ones. Further, there is no need to re-implement the algorithms, which is especially error-prone for highly sophisticated approaches.

BLUiR BLUiR is a bug localization algorithm using structured information retrieval based on code constructs, such as class and method names (Saha et al. 2013). It takes the textual information of a bug report to query the project's source code. For each source code file, its abstract syntax tree (AST) is created in order to extract class names, method names, identifiers, and comments. The Indri toolkit (Strohman et al. 2005; The Lemur Project 2020) is utilized within BLUiR for efficient indexing the extracted parts for the entire projects' code base. Next, the bug report's summary and description fields are used to query the index and retrieve similar source code files. The matching is performed using vector representations of the source code file parts and the bug report fields, i.e. summary and description. Using these vectors, a similarity calculation based on BM25 (Okapi) model (Robertson, Walker, and Beaulieu 2000) is performed. BM25 (Okapi) is a well established variant of the tf-idf model (see Eq. 2.2). It has two parameters k_1, a term weight scaling parameter, and the document normalization parameter b. The authors of BLUiR fine tuned the usually applied values from $k_1 = 1.25$, $b = 1.0$, to 1.0 and 0.3 respectively. They argue, that localization using bug reports is different from traditional text retrieval parameters need to be adjusted appropriately.

3. The author highly appreciates the descriptions of algorithms within published papers to outline the core ideas. However, often the presented algorithms are difficult re-implement in order to re-create achieved results.

Eight individual searches for all combinations of bug report fields and source code file parts are considered yielding eight separate similarity score values. The final score is than calculated by summing all individual similarities as

$$\text{score}(f, b) = \sum_{f_p} \sum_{b_f} \text{sim}(f_p, b_f) \tag{6.1}$$

, where f_p is a part of a source code file $f \in \mathcal{F}$, and b_f is a field of a bug report $b \in \mathcal{B}$. The function $\text{sim}(f_p, b_f)$ is the cosine similarity calculation for the vector representations of f_p and b_f. The $\text{score}(f, b)$ is used to rank the source code files for the bug b.

A compiled version of BLUiR is available online (Ripon Saha 2016).

AmaLgam AmaLgam is an advanced IR-based bug localization algorithm with publicly available source code (Wang 2017). It consists of three components, each calculating a suspiciousness score Susp^a, $a \in \{\text{R}, \text{S}, \text{H}\}$ for a given bug report $b_{\text{cur}} \in \mathcal{B}$. $\text{Susp}^{\text{R}}(b_{\text{cur}})$ represents *SimiScore*, which is analyzed in the next section. $\text{Susp}^{\text{S}}(b_{\text{cur}})$ is the source code structure component taken from BLUiR, and $\text{Susp}^{\text{H}}(b_{\text{cur}})$ is the history component taken from the *BugCache* algorithm (S. Kim et al. 2007). BugCache maintains a relatively short list containing the most fault-prone source code files, i.e. files, that recently were modified to fix bugs. This list is then used to predict source code files for a new bug report b_{cur}.

The composer component in AmaLgam uses two weighting factors a and b to calculate a final suspiciousness score $\text{Susp}^{\text{S,R,H}}(b_{\text{cur}})$ using the three described individual scores. $\text{Susp}^{\text{S,R,H}}$ is than used to rank the source code files.

6.2.2. Analyzing Algorithms used in Similar Bug Report Components

Similar report components utilize the knowledge about previously resolved bug reports and the thereby modified source code files, to relate them to a current, unresolved bug report. Let $B_{\text{res}} \subseteq \mathcal{B}$ be the set of all bug reports that were resolved before a current new bug report $b_{\text{cur}} \in \mathcal{B} \wedge b_{\text{cur}} \notin B_{\text{res}}$. Further, the set $F_{\text{res}} = \{\text{changed}_{\text{I}}(b) \mid b \in B_{\text{res}}\}$ denotes the set of all source code files, that were changed to resolve the bug reports in B_{res}. Using b_{cur}, B_{res}, and F_{res}, the similar issue component calculates a score for each source code file $f \in F_{\text{res}}$, which is used to created the ranked output list. It is important to note, that $F_{\text{res}} \subseteq \mathcal{F}$, i.e. the set might not correspond to all source code files in the project. It only contains those files that were part of a bug fix before b_{cur} was filed. This inherently limits the

search space of the similar issue component. For example, the component is unable to rank source code files that were never modified in order to fix a bug.

A widely applied similar issue component is *SimiScore*, which was developed by Zhou et al. for the bug localization algorithm *BugLocator* (Zhou, Zhang, and Lo 2012). *SimiScore* was later used in the more advanced algorithms *BLUiR+* (Saha et al. 2013), *BRTracer* (C. Wong et al. 2014), *AmaLgam* (Wang and Lo 2014), *AmaLgam+* (Wang and Lo 2016), and *HyLoc* (Lam et al. 2015). It is defined as follows

$$\text{SimiScore}(f, b_{\text{cur}}, B_{\text{res}}) = \sum_{\substack{b \in \{b_i | b_i \in B_{\text{res}} \\ \wedge f \in \text{changed}_{\text{I}}(b_i)\}}} \frac{\text{sim}(b, b_{\text{cur}})}{|\,\text{changed}_{\text{I}}(b)|} \tag{6.2}$$

The function $\text{sim} : \mathcal{I} \times \mathcal{I} \to \{0 \dots 1\}$ describes the textual similarity (see Eq. 2.1) between two issues.

The similar issue component *Collaborative Filtering Score (CollabScore)* was developed by Ye et al. for the bug localization approach *Learning to Rank (LR)* (Ye, Bunescu, and Liu 2014), and later reused in an improved version *LR+WE* (Ye et al. 2016). It is defined as follows

$$\text{CollabScore}(b_{\text{cur}}, f) = \text{sim}_{\text{collab}}\left(b_{\text{cur}}, \text{br}(f)\right) \qquad f \in \mathcal{F}$$

The function br returns the set of resolved bug reports for which a source code file $f \in \mathcal{F}$ was fixed before b_{cur} was filed

$$\text{br}(f) = \{b | b \in B_{\text{res}} \wedge f \in \text{changed}_{\text{I}}(b)\}$$

In order to calculate the textual similarity, the summaries of all bug reports returned by *br* are combined. CollabScore uses a different function $\text{sim}_{\text{collab}}$ than SimiScore to calculate the textual similarity. Further, it does not consider the number of source code files that have been modified to fix a bug.

6.3. Designing a Similar Issue Component - TraceScore

This section describes the TraceScore approach, a novel similar bug report component for IR-based bug localization algorithms. However, it considers all issus from the

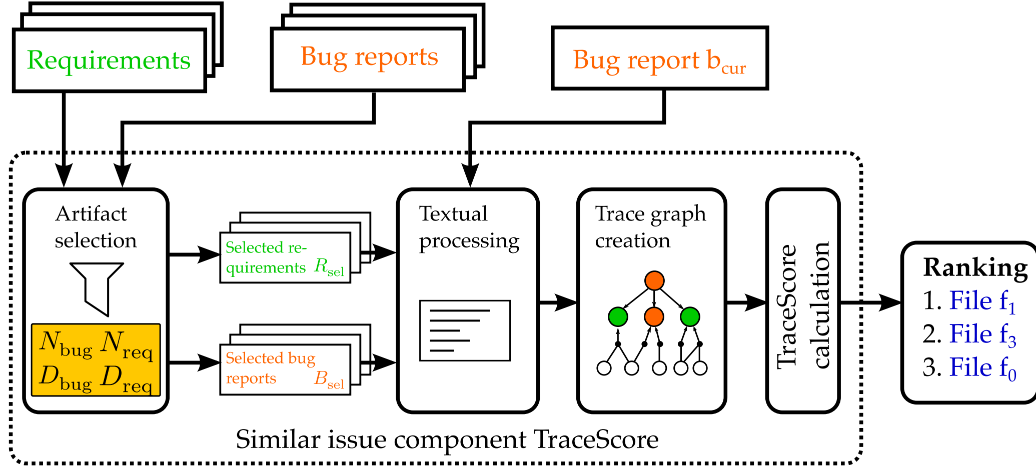


Figure 6.2.: Structure of the novel similar issue component TraceScore. The project's requirement artifacts and the current bug report b_{cur} are used as input. The output is a ranked list of source code files that are relevant for the bug report.

ITS and thus is classified as similar *issue* component. Further, the novel component uses historical project data enriched with trace data, that is currently not leveraged by existing algorithms. The TraceScore approach differs from previous approaches SimiScore and CollabScore in two points.

1. It considers *all* issue artifacts, not only bug reports, that were resolved before b_{cur} was created. Thus it also covers other development activities that modify the source code and thus may introduce bugs, such as realizing requirements or implementing features.
2. Not *all* previously resolved bug reports and issues are incorporated in the approach. Instead, the artifacts are conditionally filtered.

Overall, the approach is divided in four steps described in the following sections. The internal structure of TraceScore is shown in Figure 6.2. The inputs are bug reports and requirements that have been resolved before the current bug report b_{cur}. The output is a ranked source code file list that may contained the bug.

6.3.1. Selecting Project Artifacts

Initially TraceScore considers all previously resolved issues in order to localize a bug described in bug report b_{cur}. But, the amount of issues can become quite large, depending on developer activity and age of the project. Therefore this amount

is reduced by purposefully selecting issues. First, a filter restricts the number of changed source code files for resolving an issue. The rationale for this filter is that large source code changes in multiple files only provide small information gain on an individual source code file basis. The second filter is time based, which has been successfully applied in other studies as well (Lewis and Ou 2011). It excludes issue artifacts, that were resolved a specified time before bug report b_{cur} was filed. The reasoning for this filter is, that the maturity of a source code files increases the longer it has not been modified. The function $\text{diff}(a, b) = d$, with $a, b \in \mathcal{I}$, $d \in \mathbb{N}$ calculates the number of days d issue a was resolved before issue b was reported. However, instead of using a smoothing effect along the time axis (Lewis and Ou 2011), TraceScore applies a hard cut off. Formally, the artifact selection and filtering leads to the following two artifact sets.

$$\begin{aligned} R_{\text{sel}} &= \{r \,|\, |\text{changed}_{\text{I}}(r)| \leq N_{\text{req}} \wedge \text{diff}(r, b_{\text{cur}}) \leq D_{\text{req}}\} \quad r \in \mathcal{R} \\ B_{\text{sel}} &= \{b \,|\, |\text{changed}_{\text{I}}(b)| \leq N_{\text{bug}} \wedge \text{diff}(b, b_{\text{cur}}) \leq D_{\text{bug}}\} \quad b \in \mathcal{B} \end{aligned} \tag{6.3}$$

The parameters N_{req} and N_{bug} define an upper limit for the number of changed source code files for the respective requirement and bug report. The upper limits for the time difference is specified by parameters D_{req} and D_{req}.

6.3.2. Textual Processing

The summary and description of the issues $i \in \{b_{\text{cur}}\} \cup B_{\text{sel}} \cup R_{\text{sel}}$ are preprocessed using a standard pipeline as described in Section 2.5.1. First, stop words are removed, and combined words (CamelCase) are split. Afterwards the tokens are lowercased and stemmed. The resulting terms built the vocabulary of an issue corpus. Next all vector space representations of the issues are created like for SimiScore, i.e. using the logarithmic *tf-idf* term weighting scheme.

6.3.3. Constructing a Traceability Graph

The selected artifact sets R_{sel}, B_{sel}, the current bug report b_{cur}, and all modified source code files by these artifacts (F_{sel}) are used as nodes to construct the traceability graph. The set F_{sel} is defined as

$$F_{\text{sel}} = \Big\{ f | f \in \{\text{changed}_{\text{I}}(i) | i \in R_{\text{sel}} \cup B_{\text{sel}}\} \Big\}$$

Edges between the issue artifacts and source code files are added based on respective $\text{changed}_{\text{I}}$ sets, i.e. $\text{changed}_{\text{I}}(i, f)$ $\forall i \in R_{sel} \cup B_{\text{sel}}$, $\forall f \in F_{\text{sel}}$. Lastly, weighted edges

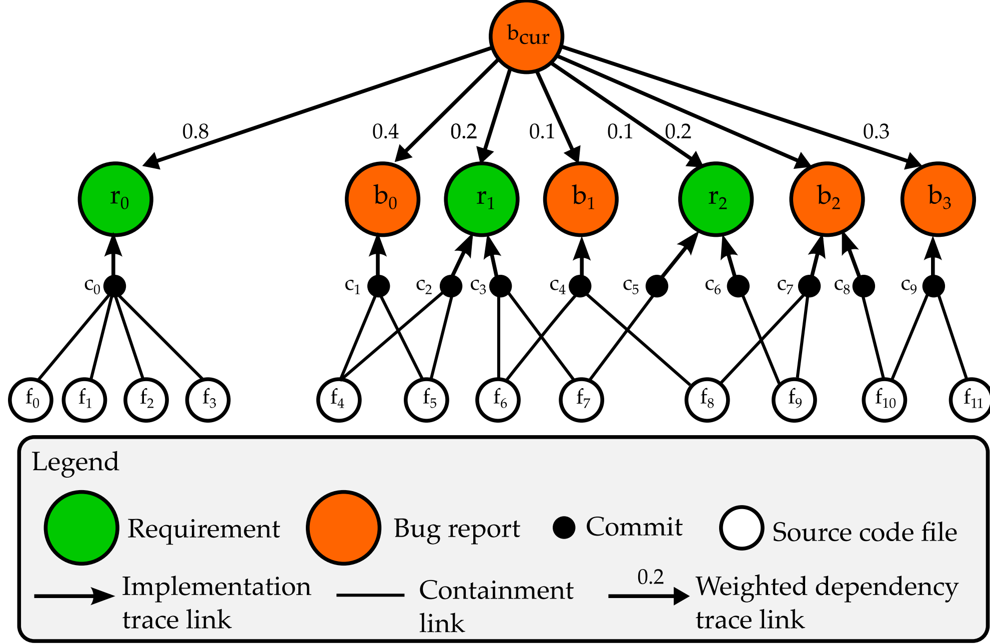


Figure 6.3.: Example traceability graph according to the defined artifact model created to recover trace links for bug report b_{cur} to source code files.

from b_{cur} to all selected issues $R_{\text{sel}} \cup B_{\text{sel}}$ are added. The edge weight is defined by the following weighting function

$$\text{weight}(i, b) = \begin{cases} 1 & \text{if } \text{isLinked}_{\text{I}}(i, b) = 1 \\ \text{sim}(i, b) & \text{otherwise} \end{cases} \qquad i, b \in \mathcal{I} \tag{6.4}$$

The resulting graph allows to trace from b_{cur} to all selected source code files in F_{sel}. An issue-to-issue edge weight is 1 if b_{cur} is linked to another issue, i.e. a developer explicitly created a trace link in the ITS between these issues. In case no such trace link exists, text based similarity calculated by sim using the same formula as SimiScore (see Eqs. 2.1, 2.2) is used to determine the edge weight.

Figure 6.3 shows an example for such a traceability graph. Its nodes are the current bug report b_{cur}, the selected requirements $R_{\text{sel}} = \{r_0, r_1, r_2\}$, the selected bug reports $B_{\text{sel}} = \{b_0, \dots, b_3\}$, and the respective source code files $F_{\text{sel}} = \{f_0, \dots, f_{11}\}$, that were changed to resolve the requirements and bug reports.

6.3.4. Analyzing the Traceability Graph

The traceability graph is used to calculate a TraceScore for each contained source code file. The formula is an evolved version of the one used by SimiScore (see Eq. 6.2). Simply applying the SimiScore formula also to requirements is not advantageous. Incorporating these new artifacts potentially extends the bug localization search space to more defective files, especially those that were never modified for a bug fix before. But this larger search space also increases the likelihood to retrieve false positives. The reason for this is the SimiScore calculation scheme which sums up ratios for each source code file $f \in \mathcal{F}$. Each ratio is derived from the textual similarity of the current bug report b_{cur} to a bug report $b \in B_{\text{res}}$ that modified f divided by the total number of source code files that were modified to resolve b. For example, the SimiScores for the source code files f_0 and f_{10} in Figure 6.3 are

$$\begin{aligned} \text{SimiScore}(f_0, b_{\text{cur}}) &= \frac{\text{sim}(r_0, b_{\text{cur}})}{4} = \frac{0.8}{4} = 0.2 \\ \text{SimiScore}(f_{10}, b_{\text{cur}}) &= \frac{\text{sim}(b_2, b_{\text{cur}})}{3} + \frac{\text{sim}(b_3, b_{\text{cur}})}{2} = \frac{0.2}{3} + \frac{0.3}{2} \approx 0.22 \end{aligned} \tag{6.5}$$

The source code file f_0 was changed to implement requirement r_0, which has a high textual similarity to b_{cur}. Thus, it is likely, that this source code file needs to be modified to resolve the bug report. However, the source code file f_{10} has a higher SimiScore. This file was changed for bug fixing multiple bug reports with far *less* textual similarity as b_{cur} and therefore it is unlikely it needs to be changed to resolve b_{cur}. But the summation (linear combination) of the ratios, which are nonlinear terms because of the underlying cosine function to calculate the similarity, vanishes the discriminating similarity terms. Thus f_{10} is ranked higher than f_0 by SimiScore neglecting the textual similarity values.

TraceScore overcomes these deficiencies based on the following assumptions.

1. A small number of source code files F_{bug} need to be changed to fix a bug:

$$F_{\text{bug}} = |\,\text{changed}_{\text{I}}(b)| = \textit{small}$$

2. Implementing requirements is more complex than resolving bug reports. Thus, on average, more source code files $F_{\text{req}} = |\,\text{changed}_{\text{I}}(r)|$, $r \in \mathcal{R}$ need to be changed:

$$F_{\text{req}} > F_{\text{bug}}$$

3. Source code files are more often changed by bug fixes than for implementing requirements:

$$
\begin{aligned}
T_{\text{bug}} &= \left|\{b | b \in \mathcal{B} \wedge f \in \text{changed}_{\text{I}}(b)\}\right| \\
T_{\text{req}} &= \left|\{r | r \in \mathcal{R} \wedge f \in \text{changed}_{\text{I}}(r)\}\right| \\
T_{\text{bug}} &> T_{\text{req}}
\end{aligned}
$$

Using these assumptions, TraceScore is defined as

$$
\text{TraceScore}(f, b_{\text{cur}}, R_{\text{sel}}, B_{\text{sel}}) = \sum_{\substack{i \in \{x | x \in R_{\text{sel}} \cup B_{\text{sel}} \\ \wedge f \in \text{changed}_{\text{I}}(x)\}}} \frac{\text{weight}(i, b_{\text{cur}})^2}{|\text{changed}_{\text{I}}(i)|} \tag{6.6}
$$

The first and second assumption affect the denominators in the formula. For example, there are two source code files f_0 and f_1, whereas f_0 is changed to implement a requirement r, and f_1 to fix a bug b. Both artifacts are connected to b_{cur} with the same edge weights, i.e. $\text{weight}(r, b_{\text{cur}}) = \text{weight}(b, b_{\text{cur}}) = w$. Using the second assumption, the added ratio for a source code file is larger, if it was changed because of a bug fix: $\frac{w^2}{F_{\text{bug}}} > \frac{w^2}{F_{\text{req}}}$.

The third assumption affects the number of summed terms. If a source code file is modified to fix multiple bugs, the ratio terms add up and, because of assumption 2, would result in similar problems of vanishing textual similarity as described for SimiScore. The effect is compensated by squaring the edge weights and pruning the traceability graph. The square function reintroduces nonlinearity by dampening low textual similarity. However, explicit established trace links with an edge weight of 1 remain unchanged. The pruning step occurs during the artifact selection (see Eq. 6.3). The parameters D_{req} and D_{bug} limit T_{req} and T_{bug}, whereas the parameters N_{req} and N_{bug} constrain F_{req} and F_{bug}.

Calculating the TraceScore for f_0 and f_{10} in Figure 6.3 results in

$$
\begin{aligned}
\text{TraceScore}(f_0, b_{\text{cur}}) &= \frac{0.8^2}{4} = 0.16 \\
\text{TraceScore}(f_{10}, b_{\text{cur}}) &= \frac{0.2^2}{3} + \frac{0.3^2}{2} \approx 0.058
\end{aligned} \tag{6.7}
$$

Contrasting SimiScore (see Eq. 6.5), TraceScore ranks f_0 higher than f_{10}, i.e. it is more likely f_0 needs to be changed to resolve b_{cur}.

The differences between Equation 6.2 and Equation 6.6 are subtle, but combined with the artifact selection, have a significant influence for achieved bug localization performance, as discussed in Section 8.

6.4. Refining the Source Code Structure Component - LuceneScore

LuceneScore is a source code structure component inspired by BLUiR. BLUiR performs multiple searches for each bug report and source code file, based on the bug reports fields and the source code file parts. Then, a final score is calculated by summing the individual search results (see Eq. 6.1). This summation is handled internally by the Indri toolkit (The Lemur Project 2020) utilized in the retrieval process. However, the toolkit is no longer actively developed, difficult to compile nowadays, and thus hard to adapt to new use cases. Therefore an evolved version of BLUiR, called LuceneScore, using the open-source project Apache Lucene™ (Apache Lucene Developers 2021) was developed for this thesis. The motivation is to support capturing the individual searches results for each source code file part. This allows to apply more sophisticated methods for integrating source code structure information rather than a simple sum as in BLUiR (see Eq. 6.1).

The internal structure of LuceneScore is shown in Figure 6.4. LuceneScore creates the abstract syntax trees for all source code files $f \in \mathcal{F}$ using the open-source `JavaParser` toolkit (JavaParser Developers 2021) in order to extract contained class names, method names, identifiers, and comments. LuceneScore performs the same text preprocessing steps, i.e. tokenization, stop word removal, and stemming, on the extracted source code file parts as described in the original BLUiR publication (Saha et al. 2013). Afterwards, the preprocessed source code parts are indexed with Apache Lucene™. Like in BLUiR the BM25 (Okapi) model is used, but the default values of $k_1 = 1.25$ and $b = 0.3$ are not changed, contrasting BLUiR. For retrieval, the summary and description of a bug report $b_{cur} \in \mathcal{B}$ are preprocessed in the same way as extracted source code file parts. Then, five individual searches are executed, each resulting in a score value. The first four scores are retrieved by searching only in the respective source code file parts, i.e. class names, method names, identifiers, and comments, yielding the scores: $\text{lucene}_{\text{class}}(f, b_{\text{cur}})$, $\text{lucene}_{\text{meth}}(f, b_{\text{cur}})$, $\text{lucene}_{\text{ident}}(f, b_{\text{cur}})$, and $\text{lucene}_{\text{comm}}(f, b_{\text{cur}})$ using the combined and tokenized summary and description of the bug report as input. The fifth retrieval is a combined search in all source code file parts *at once* yielding $\text{lucene}_{\text{comb}}(f, b_{\text{cur}})$.

To conclude, LuceneScore conceptually operates like BLUiR, but (a) does not change the commonly used default values of the BM25 model, (b) always uses the combination

of the bug reports' summary and description for retrieval, (c) is able to output five individual score values, and (d) supports direct retrieval from all source code file parts at once.

$\text{lucene}_{\text{comb}}(f, b_{\text{cur}})$ is the score to rank the retrieved source code files in case LuceneScore is used as standalone bug localization algorithm.

6.5. Utilizing TraceScore in a Bug Localization Algorithm - ABLoTS

A new algorithm *Automated bug localization algorithm using TraceScore (ABLoTS)* was created for this thesis to study the application of TraceScore component in an IR-based bug localization algorithm.

6.5.1. Internal Structure of ABLoTS

ABLoTS is based on ideas of AmaLgam (see Sec. 6.2). But, the similar issue component SimiScore of AmaLgam, i.e. Susp^{R}, is replaced with TraceScore. Additionally, the source code structure component Susp^{S} is replaced with the more detailed scores provided by LuceneScore. Lastly, the composer component, which uses a fixed weighting scheme in case of AmaLgam, is changed in favor for a *supervised classifier*. The resulting structure of ABLoTS is depicted in Figure 6.5.

Supervised learning refers to learn a model from labeled training data. This later allows to make predictions about future or unseen data. The term *supervised* means, that the labels are known. In case the labels are discrete classes, the supervised task is termed classification and a classifier predicts the class of given data points. In context of bug localization, the data points are pairs of bug reports and source code files, represented by features. There are two classes, either the respective source code file is relevant for the bug report or not.

6.5.2. The Composer Component of ABLoTS

The relationship between a bug report and a candidate source code files needs to be represented such that it can be processed by the classifier in the composer component. This leads to the creation of *feature vectors*. The components of theses vectors are the five score values calculated by LuceneScore, the TraceScore, and the BugCache score. For a given bug report b_{cur}, first LuceneScore is applied yielding a list l_{LS} of candidate source code files accompanied with five calculates score values (see Sec. 6.4).

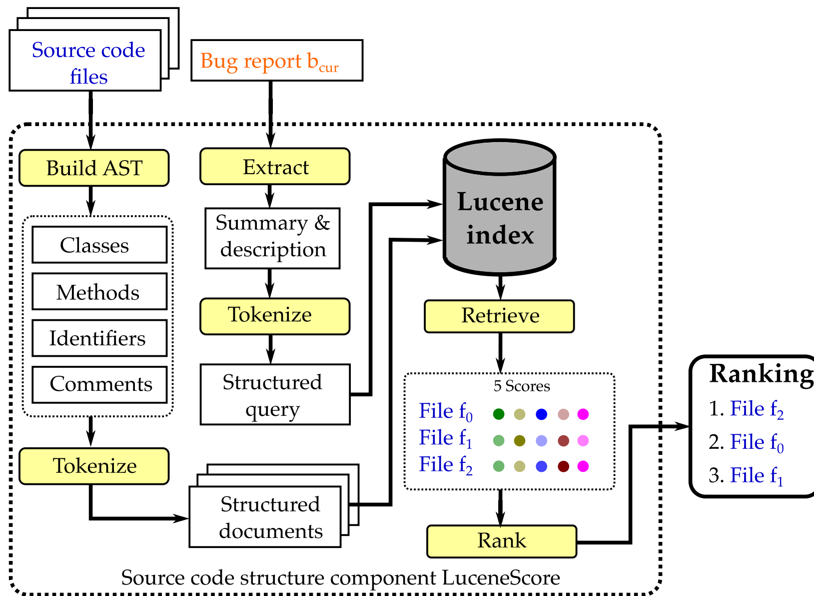


Figure 6.4.: Structure of LuceneScore, based on the works of Saha et al. (Saha et al. 2013). The projects' source code files are separately indexed by class names, method names, identifiers, and comments using Apache Lucene™. For retrieval, all textual information of a bug report b_{cur} is used as a query against the individually indexed source code parts, as well as all parts combined. This yields five score values for every retrieved source code file.

Next, TraceScore(f, b) is calculated for b, resulting in a second list l_{TS} of candidate source code files and their TraceScore values. After that, the BugCache algorithm is applied yielding the list l_{BC} of source code files accompanied by respective score BugCache(f, b). Next, the feature vectors are assembled. Therefore, for every source code file f in list l_{LS}, the five scores are taken, and the respective trace scores of f from l_{TS}, and bug cache scores from l_{BC} are collected. If no values are available, the respective score is set to 0.0. For example, based on the TraceScore algorithm, not every project's source code file has a TraceScore, because it only is "aware" of source code files that have been previously modified to resolve another issue, whereas LuceneScore operates on the whole code base. The collected score values are represented as feature vector $\mathbf{x}^{(i)} \in \mathbb{R}^{1\times 7}$ (see Eq. 6.8) having seven feature dimensions for each bug report b and source code file f

$$\mathbf{x}^{(i)} = \begin{bmatrix} \text{lucene}_{\text{class}}(f, b) \\ \text{lucene}_{\text{meth}}(f, b) \\ \text{lucene}_{\text{ident}}(f, b) \\ \text{lucene}_{\text{comm}}(f, b) \\ \text{lucene}_{\text{comb}}(f, b) \\ \text{TraceScore}(f, b) \\ \text{BugCache}(f, b) \end{bmatrix}^{T} \tag{6.8}$$

The notation $\mathbf{x}_i$ refers to the column vector for the i$^{\text{th}}$ feature dimension, and $x_i^{(j)}$ denotes the j$^{\text{th}}$ feature in this dimension. Letters in italics, i.e. $x^{(n)}$ or $x_m^{(n)}$, refer to single elements in a vector or matrix. Bold-face letters refer to vectors (lowercase) or matrices (uppercase). All feature vectors are combined into a feature matrix $\mathbf{X} \in \mathbb{R}^{n\times 7}$, whereas n depends on the retrieved source code files per bug report, which may vary. The column vector $\mathbf{y} = [y^{(0)}\, y^{(1)} \dots\, y^{(n-1)}]^T \in \mathbb{R}^{1\times n}$ with $y^{(i)} \in \{0, 1\}$ encodes, whether the source code file was modified to resolve the bug report (i.e. $y^{(i)} = 1$), which is represented by the respective feature vector $\mathbf{x}^{(i)}$. Every $\mathbf{x}^{(i)}$ and $y^{(i)}$ is an instance representing a particular combination of bug report and source code file, and each bug report is described by multiple instances (seeTab. 6.1). It is assured, that an instance for every source code file that was actually fixed to resolved a bug reports exists. In case none of ABLoTS' components suggested this file, an artificial instance with all features set to zero is created.

The feature matrix $\mathbf{X}$ and the target vector $\mathbf{y}$ are used to train a random forest classifier implemented in Python `scikit-learn` package (Buitinck et al. 2013). This type of classifier was chosen, because it has been successfully applied in other software engineering studies (Guo et al. 2004). Random forests are highly accurate and robust against noise, although they may be expensive to run on large datasets (Breiman 2001). The classifier resembles ABLoTS' composer component, and replaces the fixed weighting scheme used in AmaLgam (see Sec. 6.2). Instead, during training, the

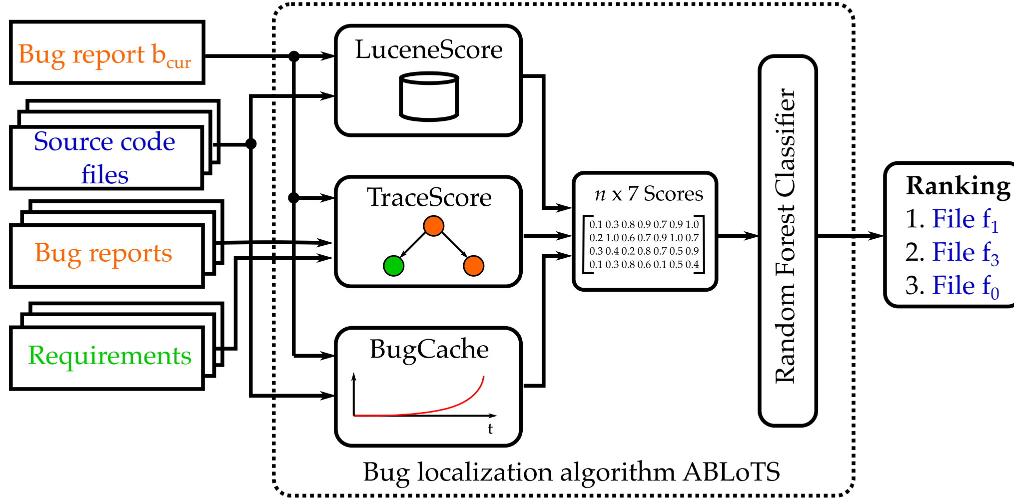


Figure 6.5.: Structure of ABLoTS bug localization algorithm utilizing LuceneScore, TraceScore, and the version history component BugCache. A random forest classifier is used as composer component to create a ranked list of source code files that need to be modified to resolve the bug report b_{cur}.

Table 6.1.: Visualization of the feature vector layout to train the composer component of ABLoTS. Each row in the table, an instance, encodes a bug report and source code file pair accompanied by the calculated feature vectors $\mathbf{x}^{(i)}$ and whether this file was fixed to resolve the bug report $(y^{(i)})$. Typically, every bug report is represented by multiple instances.

Bug report	Source code file	Feature vector	Target value
b_0	$f_?$	$\mathbf{x}^{(0)}$	$y^{(0)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
b_0	$f_?$	$\mathbf{x}^{(j)}$	$y^{(j)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
b_m	$f_?$	$\mathbf{x}^{(k)}$	$y^{(k)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
b_m	$f_?$	$\mathbf{x}^{(n-1)}$	$y^{(n-1)}$

classifier infers a function which maps the features (inputs) to outputs, i.e. whether a source code file is a candidate for fixing a bug.

Training the classifier requires that the values in a feature dimension $\mathbf{x}_j \in \mathbb{R}^{n \times 1}$ are comparable, i.e. they need to have the same scale. This is not true for the initial construction of $\mathbf{X}$. The retrieved values from LuceneScore are only valid on a per bug report basis and cannot be compared across different searches. For example, the maximum $\text{lucene}_{\text{comb}}$ score for a bug report b_0 might be orders of magnitudes larger than that for bug report b_1. These scores are only valid to rank source code files for a given bug report, not to compare source code files from different queries. Also, the magnitude of the score does not quantify its quality. The score only states, that the best match in a lucene search has the highest value, independent of the actual magnitude. The same applies to TraceScore, where the values might be arbitrarily small or large, depending on the weight distribution and number of terms in the calculated sum (see Eq. 6.6). To address the scaling issues, the feature matrix $\mathbf{X}$ is normalized per feature dimension on a *per bug report* basis. Afterwards each $x_j^{(i)}$ is in the range $0.0 \dots 1.0$.

6.6. Summary

This section described IR-based bug localization in detail. It started with the idea to integrate requirement artifacts and issue-to-issue trace links motivated by an example. Next, the internal structure of IR-based bug localization algorithms was reviewed. Here, the similar issue component was of special interest. Adding the ability to leverage requirement artifacts and trace links among them has the potential to improve the bug localization performance. Therefore two existing similar issue components were studied, current limitations discussed, and how the aforementioned artifacts can be integrated. With the gained knowledge a novel similar issue component *TraceScore* was created. Afterwards, an existing approach analyzing the source code structure was refined and *LuceneScore* created. Finally, both approaches TraceScore and LuceneScore were combined with a random forest to build the novel IR-based bug localization algorithm *ABLoTS*.

7. Automatically Augmenting Incomplete Issue-to-Commit Trace Links

Related Publications This section is based on the following publications

> **Paper III** (Rath et al. 2018): M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder "Traceability in the wild: automatically augmenting incomplete trace links" in ICSE, pages 834-845, 2018

> **Paper V** (Rath, Tomova, and Mäder 2020): Michael Rath, Mihaela T. Tomova, and Patrick Mäder "SpojitR: Intelligently link development artifacts" in International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020

The bug localization approach developed in the previous section showed, that issue artifacts and trace links can be utilized to improve the bug localization performance. It relies on trace links from issues to source code files. Manually establishing and maintaining these links requires a lot of effort. In non-regulated domains such as open-source systems this task is often perceived as prohibitively expensive. The majority of papers addressing traceability in open-source systems have focused on directly establishing trace links between issues and source code (Canfora and Cerulo 2005; Zhou, Zhang, and Lo 2012; Kuang et al. 2012; Kuang et al. 2015; Kuang et al. 2017). Therefore, a novel approach focusing on generating the missing trace links at the commit level is proposed. It also incorporates information of the project's workflow, lifecycle and stakeholder interaction, and thus not solely relies on textual similarities as most trace link recovery algorithms do. This provides traceability support within the natural context in which developers are creating trace links. The key figures of dataset shown in Table 5.2 revealed, that trace links exist in every project, but are not complete. The developers failed to create or simply forgot to add them in the first place (Bachmann and Bernstein 2009; Romo, Capiluppi, and Hall 2014). Therefore there is potential to augment the existing issue to source code trace link set for the projects. The trace links between issues and commits further provide building blocks for inferring project-wide traceability, which enables more accurate support for defect prevention (Rempel and Mäder 2017), change impact analysis, coverage analysis, and enhanced support for recommending developers for bug fixing (Anvik, Hiew, and Gail C. Murphy 2006). Thus an automated approach called *trace link set augmentation (TLSA)* based on machine learning techniques is

proposed in this section. It leverages existing issue-to-commit trace links in a project which have been created by tagging commit messages. Additionally, information of the commit process and textual similarities is used.

7.1. Analyzing Existing Issue-to-Commit Trace Links

The Tables 7.1, 7.2 show key figures of existing trace links for a subset of the SEOSS projects[1]. In particular, Table 7.1 depicts the linkage from the commit perspective, i.e. a commit is linked to exactly one bug or requirement (1:1 columns), to multiple issues (1:n column), or not linked at all. If linked, a commit is most commonly linked to exactly one issue. An exception is project Hadoop, where a larger proportion of commits are tagged with multiple issue identifiers. An example for such a commit is `04cc1d614`[2] which reads *"HADOOP-8251. Fix SecurityUtil.fetchServiceTicket after HADOOP-6941. Contributed by [developer name left out]"*. It contains two issue references and thus the commit is traced to these issues. However, this seems to be a pattern specific for project Hadoop. Interestingly, there are different ratios for unlinked commits ranging from low linked project Errai (91% of commit are not linked), up to highly linked project Zookeeper, where only 8% of commits are not linked. On average, 40% of all commits are not linked in the studied projects.

Table 7.2 shows the trace links viewed the other way around, i.e. from the issue perspective. An issue, either bug report or requirement, is linked to exactly one commit (1:1 columns), or to multiple commits (1:n columns). It is important to note, that in this specific subset of the SEOSS dataset all selected issues are linked to at least one commit, and thus no nonlinked issues exist. There is a trend, that it is more likely a bug is traced to exactly one commit. For example, in project Hive this is true for 98% of all bug reports. On the lower end is project Keycloak, where only 26% of all bug reports follow this pattern. On average 77% of all bug reports are traced to exactly one commit. A similar behavior can be seen for requirements, but lower in magnitude. Revisiting the mentioned example projects, 93% percent of requirements are linked to one commit (project Hive), and 26% for project Keycloak. On average 68% of all requirements are linked to exactly one commit.

1. The motivation for this special subset is given in the evaluation (see Sec. 8.2.1)
2. https://bit.ly/3iP6IY9

Table 7.1.: Linkage from commit perspective in (subset of) SEOSS. A commit is either linked to exactly one requirement (Req.) or bug report (1:1 columns), to multiple ones (1:n columns) or not linked at all.

Project	#Linked 1:1 Bug Reports	#Linked Req.	#Linked 1:n	#Unlinked	Unlinked Ratio
Archiva	485	796	125	2,396	0.64
Derby	2,547	2,902	287	909	0.14
Drools	1,937	2,322	156	4,164	0.49
Errai	315	222	11	5,497	0.91
Flink	1,452	1,792	70	5,912	0.64
Groovy	2,275	1,147	127	4,382	0.56
Hadoop	9,849	10,652	1,685	267	0.01
Hbase	5,092	4,880	328	622	0.06
Hibernate	2,781	2,568	151	778	0.13
Hive	4,879	3,959	183	311	0.03
Infinispan	2,908	2,952	137	2,286	0.28
Izpack	781	441	44	2,615	0.68
Jbehave	212	928	13	685	0.38
JTM	459	385	14	277	0.25
Jbpm	383	463	21	2,718	0.76
Kafka	655	647	41	649	0.33
Keycloak	1,780	1,691	131	3,253	0.48
Log4j2	843	1,273	98	4,858	0.69
Maven	868	879	50	4,074	0.70
Railo	588	202	10	1,866	0.70
Resteasy	534	446	41	1,748	0.63
Seam2	953	650	48	4,095	0.72
Spark	311	852	132	485	0.28
Switchyard	532	1,013	12	144	0.08
Teiid	1,897	2,374	303	1,082	0.20
Weld	937	943	46	2,958	0.61
Zookeeper	500	336	11	73	0.08

Table 7.2.: Linkage from issue perspective in (subset of) SEOSS. A bug report or requirement (Req.), is either linked to exactly one commit (1:1 columns), or to multiple ones (1:n columns). Please note, in this special subset all issues are linked to at least one commit.

Project	#Bug Reports	#Req.	#Linked 1:1 Bug Reports	#Linked 1:n Bug Reports	#Linked 1:1 Req.	#Linked 1:n Req.
Archiva	84	265	51	33	142	123

Table 7.2.: (continued) Linkage from issue perspective in (subset of) SEOSS. A bug report or requirement (Req.), is either linked to exactly one commit (1:1 columns), or to multiple ones (1:n columns). Please note, in this special subset all issues are linked to at least one commit.

Project	#Bug Reports	#Req.	#Linked 1:1 Bug Reports	#Linked 1:n Bug Reports	#Linked 1:1 Req.	#Linked 1:n Req.
Derby	1,782	1,302	1,380	402	768	534
Drools	1,386	731	1,162	224	379	352
Errai	233	162	207	26	122	40
Flink	1,289	1,418	1,167	122	1,163	255
Groovy	424	783	324	100	570	213
Hadoop	6,408	7,422	3,382	3,026	4,190	3,232
Hbase	4,353	4,070	4,041	312	3,422	648
Hibernate	1,604	1,336	1,112	492	852	484
Hive	4,623	3,643	4,548	75	3,393	250
Infinispan	2,274	1,761	1,946	328	1,315	446
Izpack	363	210	125	238	86	124
Jbehave	50	373	31	19	206	167
JTM	308	258	215	93	176	82
Jbpm	328	300	287	41	222	78
Kafka	594	592	577	17	523	69
Keycloak	945	799	247	698	210	589
Log4j2	488	402	382	106	221	181
Maven	151	419	124	27	336	83
Railo	300	173	270	30	153	20
Resteasy	322	173	196	126	96	77
Seam2	776	551	690	86	467	84
Spark	290	961	280	10	866	95
Switchyard	415	600	329	86	315	285
Teiid	1,406	1,293	1,123	283	696	597
Weld	623	515	492	131	344	171
Zookeeper	456	310	431	25	292	18

The goal is trying to tag each commit with at least on issue identifier, based on the observations depicted in Tab. 7.1, i.e. commits rarely tag multiple issues. Only nonlinked commits are considered in the TLSA approach. For these, there are two viable case: either an appropriate issue exists and a trace link can be established, or there is no such issue. Sometimes it is not possible or desirable to tag a commit. For example commits fixing trivial tasks are not required to be linked to any issue like commit `bf1d080` of project Groovy reading *“Fixed typo in method name (interval takes a single ending L)”*.

7.2. Motivating Example to Automatically Tag Commit Messages

Figure 7.1 shows bug report `GROOVY-5082`[3] and commit `b1bb2ab`[4] of project Groovy. Carefully reading the commit message reveals, that there is a spelling mistake of the issue identifier, because an **O** is missing. Thus the attempt of the developer to trace from the commit to the bug reports fails. This is just one example for missing trace links in a project. Arguably the most frequent case is that developers simply forget to add issue identifiers to commit messages. However, there are still potential clues, that the commit is related to the depicted bug report. The bug reports' description exhibits textual similarity with the commit message, as well as with the content of the source code file `AsmClassGenerator.java` modified by the commit. Further, the bug report was resolved on the same day as the commit was filed. Finally, the assignee of the bug report and the committer (both obfuscated in the screenshots) are the same person. Thus, the three observations provide a degree of evidence that there should be a trace link between the depicted artifacts. Therefore, an approach could leverage this information to assist the developers to automatically establish trace links by tagging the commit messages. The proposed, novel TLSA algorithm is a solution to solve this problem.

7.3. Developing a Commit Message Tagging Model

This section introduces a model, which is then used to derive an automated approach to tag commit messages. This commit message tagging model is based on the agile software development workflow were artifacts, i.e. issues, commits, and source code files are constantly created and modified over a project's lifetime. Therefore the artifact model depicted in Figure 4.2) is leveraged.

The TLSA approach considers clues from the software development process to aid the generation of trace links between issues and commits. Obviously, this process depends on time: issues are constantly created and resolved (see Fig. 2.5), and commits are submitted to the version control system. Figure 7.2 demonstrates this scenario showing two bugs reports $\{b_0, b_1\}$, four requirements $\{r_0, \ldots, r_3\}$, ten commits $\{c_0, \ldots, c_9\}$, and six source code files $\{f_0, \ldots, f_5\}$. The artifacts are ordered across the projects lifetime, e.g. requirement r_0 was created at the start of the project ($t = 0$) and resolved at time $t = 3.5$, whereas the first commit c_0 occurred at time 0.5. There are four linked issue-commit pairs $\{(r_0, c_0), (r_1, c_4), (r_3, c_8), (b_0, c_9)\}$. The

3. https://issues.apache.org/jira/browse/GROOVY-5082
4. https://bit.ly/2WcKwjw

Groovy / GROOVY-5082

Sometimes invalid inner class reference left in .class files produced for interfaces

Details

Type: Bug
Priority: Minor
Affects Version/s: 1.8.3
Component/s: class generator
Labels: None
Status: CLOSED
Resolution: Fixed
Fix Version/s: 1.8.6, 2.0-beta-3

People

Assignee:
Reporter:
Votes: 0 Vote for this issue
Watchers: 1 Start watching this issue

Dates

Created: 17/Oct/11 13:45
Updated: 01/Feb/12 11:10
Resolved: 01/Feb/12 11:10

Description

Compile this:

```
interface X {
  public String compute();
}
```

(a) Bug report `GROOVY-5082` of project Groovy.

GROVY-5082: remove synthetic interface loading helper class in case i...
…t is not used

Browse files

master
GROOVY_4_0_0_ALPHA_3 ... GROOVY_2_0_0_BETA_2

committed on 1 Feb 2012 1 parent e02ff9e commit b1bb2abfde414950238ff4d895bf5e182793500a

Showing 1 changed file with 13 additions and 2 deletions.

Unified Split

(b) Commit `b1bb2ab` with spelling mistake of issue identifier.

Figure 7.1.: Example of failed attempt to create a trace link between an issue and a commit in project Groovy because of a spelling mistake in issue identifier of the commit message (missing **O**).

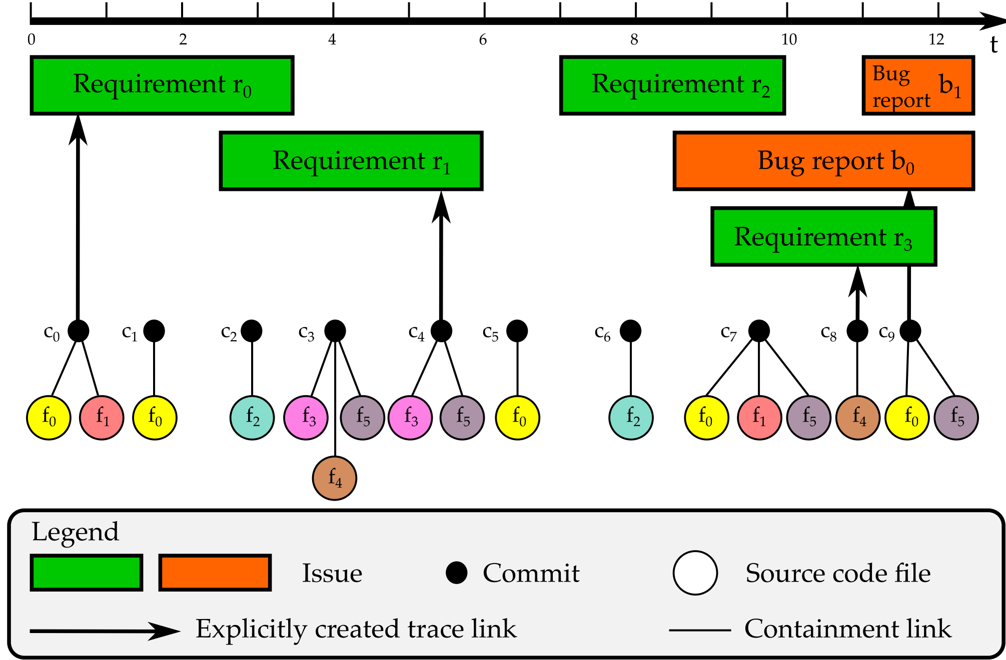


Figure 7.2.: Temporal and structural relations between issues $\mathcal{I}$, commits $\mathcal{C}$, and source code files $\mathcal{F}$ artifacts during a project's lifetime. Issues and commits are constantly created. Some commit messages are tagged with issue identifiers and thus create a trace link to the respective issue.

given examples can be formally described with the sets and functions introduced in Equation 4.1

$$\begin{aligned}
\text{created}(r_0) &= 0 \\
\text{resolved}(r_0) &= 3.5 \\
\text{committed}(c_0) &= 0.5 \\
\text{commits}(r_0) &= \{c_0\} \\
\text{commits}(r_1) &= \{c_4\} \\
\text{commits}(r_3) &= \{c_8\} \\
\text{commits}(b_0) &= \{c_9\}\,.
\end{aligned} \tag{7.1}$$

7.3.1. Analyzing the Development Process

Several relations of the development process can be stated based on the commit tagging model introduced in the previous section. The example development process

shown in Figure 7.2 contains *temporal relations*, i.e. ordering along the project's lifetime, among the artifacts. Three constraints can be derived regarding the creation of issues and commits. Further, *structural relations* can be stated, which consider the the modified source code files in a series of commits. Using these relations allows to define issue commit candidate pairs to be linked.

Temporal relations Considering a nonlinked commit $c \in \mathcal{C}$, the temporal structure imposes several constraints on possible link candidates $i \in \mathcal{I}$.

1. committed(c) < created(i): Due to causality, a commit c is not considered to be a trace link candidate for i, if c is committed before i was created. For example in Figure 7.2 the commit c_1 is not a link candidate for requirement r_1.
2. created(i) ≤ committed(c) ≤ resolved(i): This situation resembles the ordinary development workflow. Once an issue is created, the developers modify the project's source code to resolve it. Therefore commits are submitted and traced to the respective issue. Eventually the issue is resolved and no further commits for the issue occur. Thus, in Figure 7.2 the nonlinked commit c_6 is a link candidate for requirement for r_2.
3. resolved(i) < committed(c): In this case, the commit c is not considered as a link candidate for i, because it was created after i was resolved. However, this situation still might occur. One reason for this is, that the developers forgot to submit the commit before resolving the issue. Figure 7.3 shows the median time of these "late" commits for the subset of the SEOSS dataset. For example, in project Flink this value is 21 days, i.e. half of the commits filed for an issue after it was resolved occurred within three weeks. However, there are also extreme cases like commit `0d2e8b2`[5] for improvement `FLINK-3232`[6] which was filed in November 2016 nearly one year after the improvement was resolved. Another reason are unsynchronized clocks. The ITS and VCS are decentralized, independent, and unconnected systems often running on different computer systems. This also prevents strict time comparisons.

These three temporal constraints limit the number of potential issue-commit candidate pairs that may be considered for linking.

Structural relations Table 7.2 shows (1:n columns), that it is not uncommon that a series of commits are submitted to resolve an issue. The developers decompose the work into small pieces, and ideally all commits are traced to this issue. However, often only one commit in such a series is explicitly tagged with relevant issue i. Schermann et al. term these commits as *phantoms* (Schermann et al. 2015). But, all commits in the series share commonalities. First of all, they are sequential in

5. https://bit.ly/3Cqs0EA
6. https://issues.apache.org/jira/browse/FLINK-3232

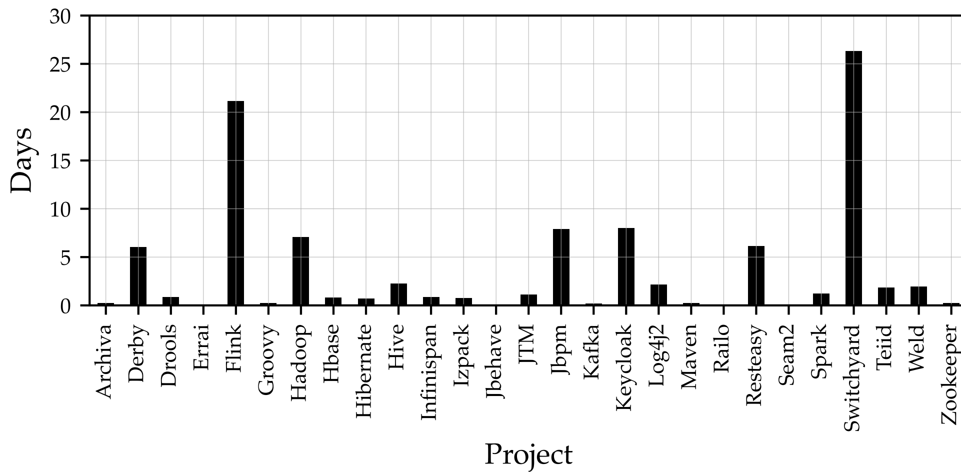


Figure 7.3.: Median time (in days) of commits linked to issues that have been resolved before the commit was submitted.

time. Next, the commits may modify a similar set of source code files as they are all purposed to resolve the same issue i. This is expressed by the function overlap

$$\text{overlap}(c_a, c_b) = \frac{|\,\text{changed}_\text{C}(c_a) \cap \text{changed}_\text{C}(c_b)|}{\max\left(|\,\text{changed}_\text{C}(c_a)|, |\,\text{changed}_\text{C}(c_b)|\right)} \tag{7.2}$$

with c_a, $c_b \in \mathcal{C}$. In Figure 7.2 the overlap of commits c_0 and c_1 is $\text{overlap}(c_0, c_1) = |\{f_0\}|/\max(1, 2) = 0.5$, whereas $\text{overlap}(c_2, c_3) = 0$. The depicted commits c_0 and c_1 have a large overlap and are temporally close, and thus c_1 may also be traced to r_0 like c_0. This situation may also occur forward in time, i.e. c_7 was committed closely before c_9, and both commits also have a high overlap. Therefore c_7 is also a possible link candidate for b_1. The average source code file overlap of consecutive commit linked to the same issue is shown in Figure 7.4. For example, for project Groovy this value is above 50%, i.e. on average consecutive commits have every second source code file in common. Sometimes all source code files are equal, e.g. for bug report `GROOVY-6094`[7] the commits `9362865`[8] and `b3686e9`[9] both modify the same three source code files `AntlrParserPlugin.java`, `ModuleNode.java`, and `ASTHelper.java`.

7. https://issues.apache.org/jira/browse/GROOVY-6094
8. https://bit.ly/3yxLFjk
9. https://bit.ly/3fG4SIt

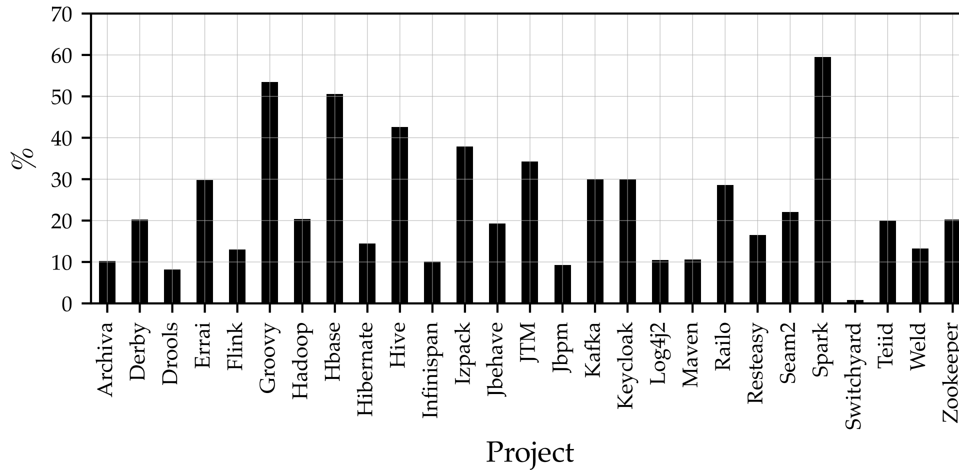


Figure 7.4.: Average source code file overlap of consecutive commits linked to the same issue for a subset of the SEOSS dataset.

7.3.2. Analyzing the Projects' Stakeholder Activities

Issue and commit artifacts both carry information about the author, which might be useful for the tagging model. The idea is to analyze the stakeholder activities and identify specific patterns. For example, consecutive commits performed by the same developer may belong together and thus should be tagged with the same issue identifier. Further, the developer who created an issue also might contribute in resolving it by modifying the source code and submitting commits. But, tracking the individual developers is difficult, because of the decoupled ITS and VCS that do not share a common stakeholder model. Each system maintains the stakeholder separately. To overcome this problem, the following approach was developed to identify identical developers in both systems. Both systems represent developers by name and login (either a nickname or an email address). With this knowledge, the first step is to collect this data. Next, names are merged when the same login is used, i.e. it is an alias for the same developer within either the ITS or VCS. Now, two lists with users and possible aliases exist and a heuristic is applied to unify the lists and thus create a unified stakeholder model. It is based on name comparison to identify identical developers in the ITS and VCS. For example, a developer having the same email address in both systems is likely to be the same person. To fully respect and protect the user privacy, all identified developers are assigned a unique number (*user id*). The function $\text{userid} : u \rightarrow \mathbb{N}$ returns this id for a user u of the ITS or VCS.

7.4. Creating a Trace Link Classifier

The models introduced in Sections 7.3.1, 7.3.2 are used to create a classifier that could identify issues associated with a commit. This classifier shall predict whether any issue-commit pair is related or not. Overall eleven features were identified for each issue-commit pair. The features fall into two categories: process-related information, and textual similarity between artifacts.

7.4.1. Deriving Process Related Features

The following nine process-related features are considered to model the relationship between issues, commits, and source code files. The features capture stakeholder-related, temporal, and structural characteristics of a candidate issue-commit pair $(i_{\text{cur}}, c_{\text{cur}}) \in \mathcal{I} \times \mathcal{C}$.

Stakeholder-related information, feat_0: This feature represents whether the identities of the assignee of i_{cur} is identical to the committer of c_{cur} (see Eq. 7.3). It is important to note, that the identity values cannot be leveraged as features, because they are specific for a project. Only using project independent features will be important for the evaluation in Section 8.7. For example, the user with userid = 3 in Proj_a is very unlikely to be identical to user with userid = 3 in Proj_b. However, feature feat_0 provides a project independent feature and still captures the relevant relationship between the two stakeholder as described in Section 7.3.2.

$$\text{feat}_0 = \begin{cases} 1 & \text{if userid(assignee of } i_{\text{cur}}) = \text{userid(assignee of } c_{\text{cur}}) \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

Temporal relations between issue and commit, feat_1: Based on temporal properties of i_{cur} and c_{cur}, the feature feat_1 represents the time difference between the issue creation and the commit time (see Eq. 7.4). This value is always positive as described in the process model. For candidate issue-commit pair (i_2, c_6) in Figure 7.2 this feature is $\text{feat}_1 = 1$.

$$\text{feat}_1 = \text{committed}(c_{\text{cur}}) - \text{created}(i_{\text{cur}}) \tag{7.4}$$

Closest linked commit, $\text{feat}_{2...4}$: To calculate these features, first the set of all linked commits to i_{cur} is determined as C_{linked} (Eq. 7.5). If not empty, the closest commit $c_{\text{closest}} \in C_{\text{linked}}$ is selected. The closest commit c_{closest} has the minimal, absolute time difference, for all $c \in C_{\text{linked}}$. Feature feat_2 captures this value. The feature $\text{feat}_3 = \text{overlap}(c_{\text{cur}}, c_{\text{closest}})$ captures the resource overlap. Lastly, the binary

feature feat_4 models whether c_{cur} and c_{closest} were committed by the same developer. In Fig. 7.2 the closest commit for c_7 is c_8. Therefore $\text{feat}_2 = |9.5 - 10| = 1.5$, and $\text{feat}_3 = 0$.

$$\begin{aligned}
C_{\text{linked}} &= \text{commits}(i_{\text{cur}}) \\
c_{\text{closest}} &= \min |(\text{committed}(c_{\text{cur}}) - \text{committed}(c)| \wedge c \in C_{\text{linked}} \\
\text{feat}_2 &= |(\text{committed}(c_{cur}) - \text{committed}(c_{closest})| \\
\text{feat}_3 &= \text{overlap}(c_{\text{cur}}, c_{\text{closest}}) \\
\text{feat}_4 &= \begin{cases} 1 & \text{if } \text{userid}(c_{\text{cur}}) = \text{userid}(c_{\text{closest}}) \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{7.5}$$

Issue related information, $\text{feat}_{5\ldots8}$: The feature feat_5 captures the total number of linked commits for i_{cur} before c_{cur} was committed (see Eq. 7.6). The set I_{unres} represents all unresolved issues at the commit time of c_{cur} and feature feat_6 captures its cardinality. In Figure 7.2 the candidate pair (b_0, c_7) yields $I_{\text{unres}} = \{b_0, r_2, r_3\}$ and thus $\text{feat}_6 = 3$. The feature feat_7 represents the number of unresolved issues for the committer of c_{cur} at the point in time of the commit. At last, feature feat_8 captures the issue type of i_{cur}.

$$\begin{aligned}
\text{feat}_5 &= |\{c | c \in \text{commits}(i_{\text{cur}}) \wedge \text{committed}(c) < \text{committed}(c_{\text{cur}})\}| \\
I_{\text{unres}} &= \{i |\, \text{created}(i) \leq \text{committed}(c_{\text{cur}}) \leq \text{resolved}(i) \wedge i \in \mathcal{I}\} \\
\text{feat}_6 &= |I_{\text{unres}}| \\
\text{feat}_7 &= |\{i |\, \text{userid}(\text{assignee of } i) = \text{userid}(\text{committer of } c_{\text{cur}}) \wedge i \in I_{\text{unres}}\}| \\
\text{feat}_8 &= \begin{cases} 1 & \text{if } i_{\text{cur}} \in \mathcal{B} \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{7.6}$$

7.4.2. Deriving Textual Similarity Features

Next to the process related features, two features represent textual between issues, commit messages, and source code.

Textual similarity of a commit and an issue, feat_9: The feature feat_9 is the textual similarity between the commit message of c_{cur} and the textual contents of i_{cur}, i.e. the summary and description. The previously described cosine similarity sim with logarithmic tf-idf scheme is leveraged (see Eq. 2.1).

Textual similarity of committed source code files and an issue, feat_{10}: This feature captures the textual similarity between source code and the textual contents

of i_{cur}. Thus, the combined summary and description of i_{cur} is used as query against the source code files modified by c_{cur}, i.e. $\text{changed}_{\text{C}}(c_{\text{cur}})$. This is performed using LuceneScore (see Sec. 6.4) and the value of $\text{lucene}_{\text{comb}}$ of the topmost result, i.e. the one with the highest similarity, is assigned to feature feat_{10}.

7.4.3. Creating Feature Vectors

The trace link prediction problem is formulated as a classification task that predicts for each issue-commit combination whether it is traced or not. Therefore, the aforementioned features are calculated for a given project as follows. First, the set P of candidate issue-commit pairs is constructed

$$P = \{(i, c) |\, \text{isCandidate}(i, c)\} \quad c \in \mathcal{C}, i \in \mathcal{I} \tag{7.7}$$

with the predicate isCandidate defined as

$$\begin{aligned} \text{isCandidate}(i, c) = \text{created}(i) \leq \text{committed}(c) \\ \wedge \text{committed}(c) \leq \text{resolved}(i) + \epsilon \end{aligned} \tag{7.8}$$

based on observations made in the process model in Section 7.3.1. The predicate models the temporal relations and limits the number of possible pairs according to causality. No strict time comparison is possible. The parameter ϵ allows to break strict temporal causality. Ideally, source code modifications (commits) occur before the respective issue is resolved. However, an empirical analysis of the SEOSS dataset showed, that this is not true and sometimes tagged commits were created after issue resolution. This situation is modelled using ϵ and thus commits with this range after issue resolution are still considered as commit candidates. 50% of the late commits in the SEOSS dataset occurred within less than two days on a project basis (see Fig. 7.3) and thus ϵ was set to 36 hours.

It is important to note, that there may be multiple candidate commits for a given issue, which is not unusual as shown in Table 7.2 (1:n columns). Further, a commit may be candidate for multiple issues. The exception is, that an issue-commit pair $(i, c) \in P$ represents an existing trace link, i.,e. and $\text{isLinked}(i, c) = 1$. In this case, the commit c is not considered as candidate for other issues, because the scenario where a commit is tagged with multiple issues ids is rather rare (see Tab. 7.1). The issue-commit pairs for artifacts in Figure 7.2 are shown in Equation 7.9. The blue highlighted ones represent pairs with an established trace link, e.g. (r_0, c_0). The commit c_2 is candidate for two requirements r_0, r_1, and there is no pair containing bug report b_1.

Table 7.3.: Visualization of the feature vector layout to train the TLSA classifier. Each row in the table, an instance, encodes an issue-commit pair accompanied by the calculated features $\mathbf{x}^{(i)}$ and whether the two artifacts are linked ($y^{(i)}$).

Issue	Commit	Feature vector	Target value
i_0	c_0	$\mathbf{x}^{(0)}$	$y^{(0)}$
i_0	c_1	$\mathbf{x}^{(1)}$	$y^{(1)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$i_?$	$c_?$	$\mathbf{x}^{(n-1)}$	$y^{(n-1)}$

$$
\begin{aligned}
&(r_0, c_0), (r_0, c_1), (r_0, c_2) \\
&(r_1, c_2), (r_1, c_3), (r_1, c_4), (r_1, c_5) \\
&(r_2, c_6), (r_2, c_7) \\
&(r_3, c_7), (r_3, c_8) \\
&(b_0, c_7), (b_0, c_9)
\end{aligned}
\tag{7.9}
$$

Next, the 11 features are calculated for each issue-commit pair and organized as a feature vector $\mathbf{x}^{(i)}$ similarly as for the ABLoTS composer component (see Sec. 6.5.2). All vectors are combined in feature matrix $\mathbf{X} \in \mathsf{R}^{n\times 11}$. The elements of the target vector $\mathbf{y} = [y^{(0)}\, y^{(1)}, \ldots\, y^{(n-1)}]^T$ with $y^{(i)} \in \{0, 1\}$ encodes isLinked(i, c), i.e. whether a trace link exists between the artifacts. $\mathbf{x}^{(i)}$ and $y^{(i)}$ represent an instance for the i$^{\text{th}}$ issue-commit pair. All instances can be visualized as shown in Table 7.3.

Like for the ABLoTS composer component (see Sec. 6.5.2), a random forest classifier was chosen with the same reasoning, and this type of classifier has been successfully used in a similar use case (Le et al. 2015). It is trained by directly using $\mathbf{X}$ and $\mathbf{y}$, i.e. without the need for feature scaling. The feature dimensions either encode boolean values represented as $\{0, 1\}$ (boolean decisions), numerical values that are inherently limited to the range $0 \ldots 1$ (textual similarities, and overlap), or numerical values representing time differences. Thus, individual elements $x_i^{(j)}$ and $x_i^{(j+1)}$ of feature dimension i are already on the same scale.

7.5. Summary

This section motivated that integrating requirement artifacts and trace links to bug localization algorithms can be advantageous to improve bug localization performance. However, the analysis of the 33 projects of SEOSS dataset showed, that often the

required trace links between issue and commit artifacts are missing. The developers failed or simply forgot to create trace links during the development workflow. Thus, the typical development workflow was analyzed to exploit possibilities to automatically add issue-to-commit trace links. Therefore a process model, a temporal model, and a stakeholder model were derived from the developers workflow. The models' capabilities were captured in 11 features, leading to the design of the *Trace Link Set Augmentation* approach based on machine learning techniques. This binary classifier takes issue-to-commit pairs and predicts, whether a trace link between these two artifacts should be established or not.

8. Evaluation

This section evaluates the algorithms introduced in the previous sections. Hereby an adapted subset of projects from the SEOSS dataset (see Sec. 5.4) is used. Overall five different experiments are described and conducted.

8.1. Research Questions

The evaluation seeks to answer the following six research questions. For each question, except for the last one, a dedicated experiment is performed.

RQ-1: Effectiveness of the TraceScore component How effective is the proposed approach for bug report to source code trace link recovery? The question answers whether the ideas to create TraceScore result in an improved similar issue component. Therefore its performance is compared with two previous work similar issue components CollabFilter and SimiScore.

RQ-2: Impact of TraceScore Parameterization What is the impact of the four TraceScore parameters, i.e. utilization of requirement artifacts, trace links, and filtering of historical artifact data? Here, the tunable parameters D_{bug}, D_{req}, N_{bug}, and N_{req} of TraceScore are evaluated to provide a baseline configuration for the component.

RQ-3: Effectiveness of an IR-based bug localization algorithm using TraceScore Do IR-based bug localization algorithms benefit from TraceScore? TraceScore is just one component of a bug localization algorithm. This research question answers its' usefulness when plugged into the novel multi-component bug localization algorithm ABLoTS. The performance is compared with existing algorithms previous work BLUiR, AmaLgam, and the novel algorithm LuceneScore.

RQ-4: Effectiveness of trace link set augmentation How effective is the developed TLSA approach? Manual trace link creation is tedious, so here the approach to automatically create issue-to-commit trace links is evaluated. This research question answers to what extend this automatic link augmentation is possible and the quality of the created links.

RQ-5: Effectiveness of IR-Based Bug Localization Algorithms on Projects with Augmented Trace Link Sets What is the performance of four IR base bug localization algorithms on projects with augmented issue-to-commit trace links? Assuming a project with incomplete issue-to-commit trace links, which has been augmented to (re-)create missing links. This question answers if IR-based bug localization is still possible on such a project, and what performance measures can be achieved. Here the same algorithms as for experiment III (RQ-3) are evaluated.

RQ-6: Limitations of Studied Approaches The last research question discusses the limitations of the developed algorithms TraceScore, LuceneScore, ABLoTS and TLSA. Further, limitations of applying them on a project is investigated, especially their interplay.

8.2. Introducing the Evaluation Datasets

Several datasets are used for evaluation. They all are based on the published SEOSS dataset (see Sec. 5.4), and address different evaluation aspects. This section motivates, defines and explains the created datasets. Figure 8.1 depicts the overall creation process.

8.2.1. Creating a Gold Standard Dataset GS

This dataset is a curated subset of the SEOSS dataset and serves as gold standard in the evaluation. Therefore three constraints are applied to build it. This is visualized in the top two boxes in Figure 8.1.

First step (curating project) SEOSS does not contain any source code and just records the git repository URL and hash value of the latest analyzed git commit hash in a meta table. For the three projects Axis2, HornetQ, and Pig the URLs were no longer valid or commit hashes could no longer be retrieved when writing this thesis. Further, the projects Cassandra, Lucene, and Wildfly are too complex in terms of number of issues, commits and their temporal coherence, that the calculation of the features for TLSA took too long[1]. Thus, these six projects were excluded and only 27 projects remain and are used for the evaluation, i.e.

$$\mathsf{GS}_{\text{project-names}} = \mathsf{SEOSS}_{\text{project-names}} \setminus \{\text{Axis2}, \text{Cassandra}, \text{HornetQ}, \text{Lucene}, \text{Pig}, \text{Wildfly}\}$$

1. The feature calculation did not finish for each of these projects within a day.

, where $\mathsf{GS}_{\text{project-names}}$ is the set of project names (see also Eq. 5.1).

Second step (filtering issues) SEOSS is a snapshot of all project artifacts. Thus, contained issues may be in any state of their workflow. However, the studied concepts require a well defined state for issue artifacts, i.e. they have to represent finished work. The selected issue artifacts need to have the resolution set to FIXED or DONE and the status has to be RESOLVED or CLOSED (see Tab. 2.1). Thus it can be assumed, that all necessary source code file modifications have been made. This condition is implemented by the function $\text{finished}(i) \leftarrow \{1,0\}$ with $i \in \mathcal{I}$. Additionally, a trace link to at least one commit containing at least one source code file needs to exist for each issue artifact. Without this, it is impossible to figure out where in the code base a requirement was implemented or a bug report resolved. These constraints are captured by the already introduced function $\text{changed}_{\text{I}}(i)$, and $\text{hasSCF}(c) \rightarrow \{1,0\}$ (read: *has source code files*) with $i \in \mathcal{I}$, $c \in \mathcal{C}$. Commits that contain only none source code files, such as ones updating the documentation or modifying the projects' build system are not considered, because these are irrelevant for bug localization. Thus, the artifact selection for the curated projects is defined as:

$$
\begin{aligned}
p &\in \mathsf{GS}_{\text{project-names}} \\
\text{Proj}_p &= \left(I_p, C_p, F_p\right) \\
I_p &= \left\{i | i \in I_p^{(\text{seoss})} \wedge c \in C_p^{(\text{seoss})} \wedge \text{finished}(i) \wedge \text{isLinked}(i,c)\right\} \subseteq I_p^{(\text{seoss})} \\
C_p &= \left\{c | c \in C_p^{(\text{seoss})} \wedge i \in I_p \wedge \text{isLinked}(i,c) \wedge \text{hasSCF}(c)\right\} \subseteq C_p^{(\text{seoss})}
\end{aligned}
$$

Third step (filtering source code files). From all commits only those changing source code files are considered. Thus, for mixed commits, i.e. commits containing modifications of different file types like source code files and e.g. project build system configurations (Makefiles etc.), only the source code files are relevant. The source code files are identified by file extension, defined as function $\text{isSCF}(f) \rightarrow \{1,0\}$ (read: *is source code file*), with $f \in \mathcal{F}$. This is modeled as

$$
F_p = \left\{f | f \in F_p^{(\text{seoss})} \wedge \text{isSCF}(f) \wedge f \in \text{changed}_{\text{C}}(c) \wedge c \in C_p\right\} \subseteq F_p^{(\text{seoss})}
$$

Combined, the three constraints filter the SEOSS dataset, first on project level, and than for each remaining project on an artifact level. For example, applying the introduced formulas on project $p = \text{Derby} \in \mathsf{GS}_{\text{project-names}}$ results in

$$
\begin{aligned}
\mathrm{Proj}_{\mathrm{Derby}} &= \left(I_{\mathrm{Derby}}, C_{\mathrm{Derby}}, F_{\mathrm{Derby}}\right) \\
I_{\mathrm{Derby}} &\subseteq I^{(\mathrm{seoss})}_{\mathrm{Derby}} \qquad \text{i.e. filtered issues} \\
C_{\mathrm{Derby}} &\subseteq C^{(\mathrm{seoss})}_{\mathrm{Derby}} \qquad \text{i.e. filtered commits} \\
F_{\mathrm{Derby}} &\subseteq F^{(\mathrm{seoss})}_{\mathrm{Derby}} \qquad \text{i.e. filtered source code files}
\end{aligned}
$$

Eventually, GS is the set of all projects defined as

$$
\begin{aligned}
p &\in \mathsf{GS}_{\text{project-names}} \\
\mathsf{GS} &= \bigcup_{p} \mathrm{Proj}_{p}
\end{aligned}
\quad . \tag{8.1}
$$

It is important to highlight, that GS is *trusted*, and therefore used as *gold standard*. All source code files modified by the issues are treated as correct, i.e. have been intentionally modified by the developers to implement the respective requirement, or resolve a bug report. This assumption may not be correct, such that some source code files may have been modified during a commit although it was not necessary (e.g. correcting a spelling mistake in source code comments). Automatic identification of such changes is a hard task and would require manual inspection. This inspection is infeasible, because of the shear amount of commits, and the lack of required expert knowledge, i.e. to detect if a source code change was indeed necessary, but is irrelevant for the respective issue.

Table 8.1 provides an overview of GS. It shows key figures such as the time period, the contained bug reports, and requirements for each of the 27 included projects. For example, project Hadoop contains the most artifacts with 6,408 bug reports, and 7,422 requirements. Further, the median, mean, and max number of modified source code files for bug reports and requirements is shown. Please note, the minimum number of modified source code files is one, as assured by the curating process.

The golden standard GS used to answer the research questions **RQ-1**, **RQ-2**, and **RQ-3**, whereas for the remaining research questions additional datasets are used. Other statistics of GS, especially the poor existing issue-to-commit linkage, is shown in Table 7.2, which motivated the creation of the TLSA approach. It is important to emphasize, that Table 7.1 *does not* show GS. This table uses I_p, but also depicts nonlinked commits taken from SEOSS, which are per design (constraint two) not contained in GS.

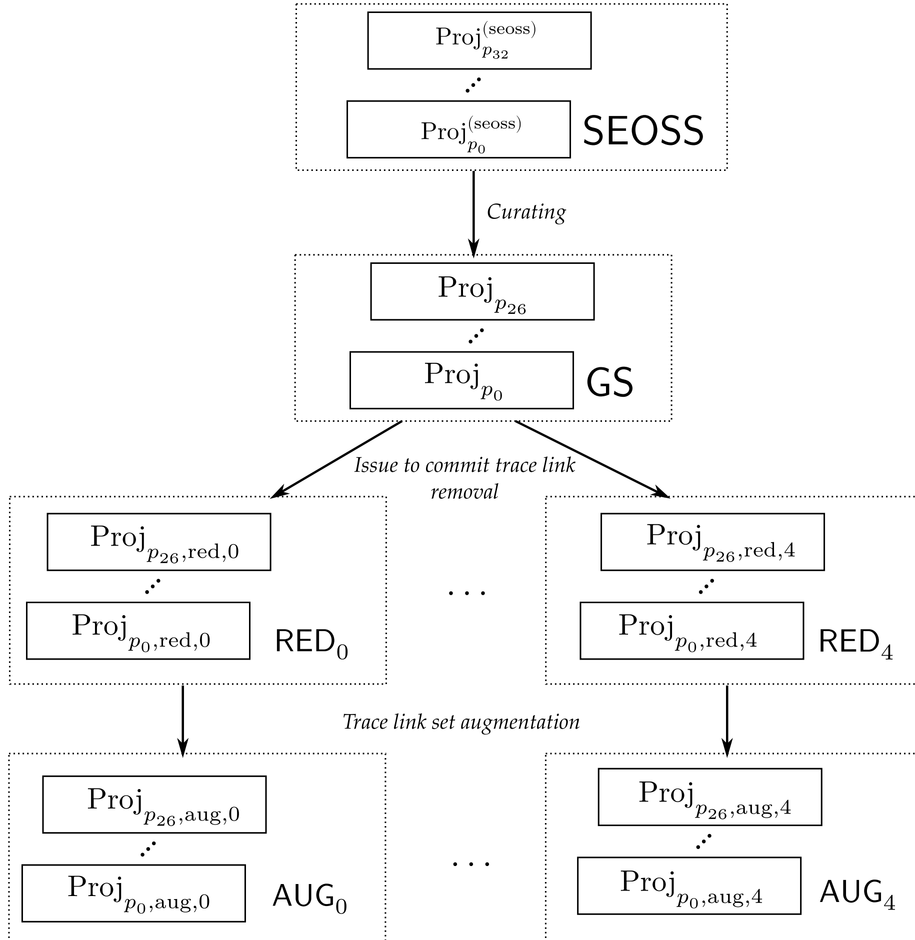


Figure 8.1.: Creation process for the datasets based on SEOSS used for evaluation. The gold standard dataset GS is created by filtering projects and artifacts from SEOSS. Next, five datasets RED_i, $i \in \{0, \dots, 4\}$ are derived from GS by removing issue-to-commit trace links. These are than processed by TLSA algorithm to construct datasets AUG_i with augmented trace link sets.

Table 8.1.: Key figures of studied projects in the curated dataset GS with project lifetime, number of bug report, requirements, trace links among them, and statistics about the number of modified source code files per issue type. Each bug report and requirement modifies at least one source code file, according to the dataset creation constraints.

Project	Time Period	#Bug Reports	#Changed Source Code Files per Bug Report			#Requirements	#Changed Source Code Files per Req.			#Trace Links
			median	mean	max		median	mean	max	
Archiva	2005-03 - 2017-08	84	2	9.0	719	265	5	14.9	526	207
Derby	2004-09 - 2017-09	1,782	2	5.5	1,919	1,302	3	10.5	674	1,997
Drools	2005-12 - 2017-11	1,386	3	6.3	168	731	9	38.1	1,835	234
Errai	2009-12 - 2017-11	233	3	5.1	75	162	6	13.4	231	18
Flink	2014-06 - 2017-11	1,289	2	5.4	128	1,418	7	17.8	490	463
Groovy	2003-09 - 2017-05	424	1	2.0	40	783	1	3.3	482	458
Hadoop	2006-05 - 2017-11	6,408	2	3.4	312	7,422	4	13.1	8,913	10,303
Hbase	2007-05 - 2017-11	4,353	2	4.4	754	4,070	3	13.2	2,005	3,509
Hibernate	2004-07 - 2017-11	1,604	3	7.2	1,213	1,336	5	37.0	7,540	1,049
Hive	2008-09 - 2017-11	4,623	2	4.0	359	3,643	3	12.7	3,055	4,177
Infinispan	2009-03 - 2017-11	2,274	3	5.3	167	1,761	6	21.6	2,851	1,037
Izpack	2009-01 - 2017-10	363	2	4.4	69	210	5	13.5	160	58
Jbehave	2008-11 - 2017-11	50	2	3.0	31	373	4	10.4	186	2
JTM	2009-01 - 2017-11	308	2	4.2	130	258	4	14.5	301	374
Jbpm	2010-01 - 2017-11	328	3	5.2	97	300	5	26.2	2,192	495
Kafka	2011-07 - 2017-11	594	2	4.4	69	592	6	12.4	130	682
Keycloak	2013-07 - 2017-11	945	2	7.5	324	799	6	20.9	2,326	592
Log4j2	2008-12 - 2017-11	488	2	5.2	230	402	4	16.7	717	261
Maven	2004-06 - 2017-10	151	2	3.4	77	419	2	8.6	396	560
Railo	2008-07 - 2013-12	300	2	8.1	721	173	4	7.3	140	17
Resteasy	2008-03 - 2017-11	322	3	6.0	142	173	5	56.6	5,218	145

Table 8.1.: (continued) Key figures of studied projects in the curated dataset GS with project lifetime, number of bug report, requirements, trace links among them, and statistics about the number of modified source code files per issue type. Each bug report and requirement modifies at least one source code file, according to the dataset creation constraints.

Project	Time Period	#Bug Reports	#Changed Source Code Files per Bug Report			#Re-quire-ments	#Changed Source Code Files per Req.			#Trace Links
			median	mean	max		median	mean	max	
Seam2	2005-08 - 2014-03	776	1	2.4	47	551	2	9.6	1,700	312
Spark	2011-11 - 2017-11	290	2	3.1	45	961	2	5.3	173	3,699
Switchyard	2010-11 - 2017-02	415	2	64.4	1,818	600	9	42.0	2,076	483
Teiid	2004-04 - 2017-11	1,406	3	6.2	325	1,293	6	28.2	3,616	396
Weld	2008-06 - 2017-11	623	4	7.2	570	515	7	18.6	1,361	328
Zookeeper	2008-06 - 2017-11	456	2	3.3	36	310	3	11.6	695	304

8.2.2. Creating Datasets with Reduced Trace Link Sets RED_i

The TLSA algorithm described in Section 7 is designed to enhance the existing trace links set of issues $\mathcal{I}$ to commits $\mathcal{C}$. Applying this task to GS *may* and *should* create new issue-to-commit trace links. But, the evaluation of the correctness of these traces is similar as proofing the correctness of trace links in GS: it is hard to automate, has do be done by expert human users (ideally the projects' developers), and would be very time consuming. Thus a different approach is applied. Instead of creating new trace links, the existing set of issue-to-commit trace links is reduced by randomly removing links. Then, the TLSA classifier is trained and applied to this reduced issue-to-commit link set to augment the missing trace links. Their correctness is then validated against the set of trace links found in GS, which per definition are treated to be correct. Thus, a perfect TLSA classifier would recreate GS out of the projects with randomly removed issue-to-commit trace links.

The creation of the reduced issue-to-commit link set is done automatically and takes care of certain constraints.

First constraint (trace link removal rate) Table 7.1 reveals, the average number of nonlinked commits across all projects is about 40% in SEOSS dataset. Thus the developers failed or forgot to tag two out of five commits with an issue identifier. Per design, GS does not contain any nonlinked commits, i.e. every issue and commit is part of a trace link. Therefore, following the original developer behavior from the SEOSS projects, 40% of the existing trace links in GS are removed. This models the original trace link failure rate as found in the projects.

Second constraint (trace link distribution) The TSLA classifier needs to be trained on a part of issue-to-commit trace links of a project (training data) and then applied on the remaining ones (testing data). The details of the process are discussed in Section 8.6.1. Randomly removing trace links could lead to undesired effects. For example, when splitting the project data in training and testing data after the link removal, there is the chance that more (in a percentage sense) trace links are removed in either set leading to an unequal distribution. Additionally, more trace links between requirements and commits could have been removed than between bug reports and commits. To ensure an equal distribution for all these cases, the 40% removal rate is individually applied to training and testing data, and within these splits individually for bug reports to commit trace links and requirement to commit trace links.

Third constraint (trace link preservation) The process model in the TLSA approach contains structural features between commits (linked and nonlinked) to issues (see Sec. 7.4). A pure random removal of trace links could lead to a situation, that an issue is no longer linked to any commit and thus no longer fits the process

model, because the calculation of structural features is infeasible. To prevent this, at least one trace link to a commit needs to remain for each issue artifact. This is always possible, because during creation of the GS only issues having one or more trace links to commits were chosen (see Eq. 8.1).

The random removal of trace links may introduce a bias. In an (unlikely) circumstance, simply the first (in a temporal sense, i.e. the start of a project's lifecycle) 40% of issue-to-commit links are removed. To mitigate this threat, the removal is repeated five times, which creates multiple issue-to-commit trace link sets per project (see the middle part of Fig. 8.1). Each removal builds a separate dataset denoted by index i. Formally, the reduced datasets are derived from GS by removing issue-to-commit links. Removing commits also removes source code files that were only contained in these commits. However, the set of issue artifacts remains unchanged (see Eq. 8.2):

$$\begin{aligned} p &\in \mathsf{GS}_{\text{project-names}} \\ i &\in \{0, \dots, 4\} \\ \text{Proj}_{p,\text{red},i} &= \left(I_p, C_{p,\text{red},i} \subset C_p, F_{p,\text{red},i} \subseteq F_p\right) \\ \mathsf{RED}_i &= \bigcup_p \text{Proj}_{p,\text{red},i} \end{aligned} \tag{8.2}$$

For example $\text{Proj}_{\text{Derby,red},0} \in \mathsf{RED}_0$ is project Derby with the first randomly reduced trace link set, $\text{Proj}_{\text{Derby,red},1}$ the second reduced trace link set and so on. The five RED_i datasets are used for answering **RQ-4** studying the performance of the TLSA algorithm.

8.2.3. Augmenting Trace Link Sets to Create Datasets AUG_i

The last five datasets are created by TLSA algorithm. They contain all issue-to-commit trace links of the reduced datasets, along with augmented ones. If TLSA is perfect, it would recreate GS for each reduced dataset RED_i, i.e. all previously removed trace links would be correctly recovered. However, it is expected that trace links between issues and commits are missing (false negatives), and some incorrect trace links are added (false positives). Since trace link removal is repeated five times, one augmented dataset AUG_i exists for each reduced one denoted by index i (see Eq. 8.3, and bottom of Fig. 8.1).

$$
\begin{aligned}
p &\in \mathsf{GS}_{\text{project-names}} \\
i &\in \{0, \dots, 4\} \\
\mathrm{Proj}_{p,\mathrm{aug},i} &= \left(I_p, C_{p,\mathrm{aug},i} \subseteq C_p^{(\mathrm{seoss})}, F_{p,\mathrm{aug},i} \subseteq F_p^{(\mathrm{seoss})}\right) \\
\mathsf{AUG}_i &= \bigcup_p \mathrm{Proj}_{p,\mathrm{aug},i}
\end{aligned}
\tag{8.3}
$$

For example $\mathrm{Proj}_{\mathrm{Derby},\mathrm{aug},0} \in \mathsf{AUG}_0$ is project Derby with an augmented trace link set created by TLSA algorithm applied on $\mathrm{Proj}_{\mathrm{Derby},\mathrm{red},0}$, i.e. the first randomly reduced variant. The definition of $C_{p,\mathrm{aug},i}$ is important, because *all* commits from the respective original project $\mathrm{Proj}_p^{(\mathrm{seoss})}$ are considered, and not only the linked ones from GS. Thus, the TLSA algorithm considers all commits, i.e. the ones were a trace link was removed from RED_i and those which were never linked to an issue from $\mathrm{Proj}_p^{(\mathrm{seoss})}$. This is much more challenging than just considering the removed ones, but also more realistic: all untagged commits are eligible candidates for trace link creation. The augmented datasets are used to evaluate the performance of TLSA algorithm in **RQ-4**, as well as to perform bug localization when answering **RQ-5**.

8.3. Evaluation Metrics

This section describes common evaluation metrics. This includes ones used for bug localization performance like Top@k, MAP, and MRR (C. Wong et al. 2014; Saha et al. 2013; Wang and Lo 2014, 2016). Further, common information retrieval metrics used to evaluate the TLSA classifier like precision and recall are introduced. Lastly, the concept of effect sizes to quantify differences between measures is described.

8.3.1. Top@k

Top@k, sometimes also called Hit@k, measures the percentage of bug reports for which at least one correct source code file is among the top k ranked files. Thus, a given bug report b is considered successfully located, if at least one correct source code file is in the k highest ranked files. The larger the value of Top@k, the better the performance of an approach.

8.3.2. Mean Average Precision (MAP)

A commonly used measure to evaluate ranking approaches is mean average precision (MAP) (Manning and Schütze 2001). It provides a single measure of quality across multiple query results. The *average precision (AP)* for each query is calculated as

$$AP = \sum_{k=1}^{M} \frac{\mathrm{P}(k) \times \mathrm{pos}(k)}{M_{\mathrm{relevant}}}$$

where k is the rank, i.e. the position in the ranked query list starting at 1. The parameter M is the number of retrieved source code files and M_{relevant} the number of relevant source code files for the given bug report. The binary function $\mathrm{pos}(k)$ returns 1, if the source code file at rank k is relevant, i.e. indeed contained a bug, and 0 otherwise. Last, $\mathrm{P}(k)$ specifies the precision at rank k. AP favors recall over precision and thus is suited for bug localization. Here, the developers have to scan through the ranked source code file list created for a bug report to identify the relevant files. Ideally the list is short and contains the most relevant files on top. MAP is the mean value across all AP scores.

Whereas the measures Top@k and RR focus on the rank of the first correctly localized source code files, MAP considers all correctly localized files. For example, a ranked result list with one correct source code file on rank five has an AP of 0.2. In case there is another correct source code file on rank six, the AP increases to 0.26, but the other metrics Top@k and RR remain unchanged. Therefore MAP is the *primary metric* throughout this thesis to judge the performance of a bug localization algorithm.

8.3.3. Mean Reciprocal Rank

The *reciprocal rank (RR)* is defined as the reciprocal of the rank of the first relevant source code file in the retrieved ranked list of source code files for a given bug report. RR emphasizes on high precision contrary to average precision. Therefore relevant files should be at the beginning of the list, ideally at the first position. The mean reciprocal rank (MRR) is the average across all queries (Voorhees 1999). MRR is suitable for developers to identify only one (the most relevant) source code file.

8.3.4. Cliff's delta

Cliff's delta $|\delta|$ is used to calculate effect sizes when comparing evaluation metrics to quantitatively measure the difference in magnitude (Cliff 1993). For example, when comparing MAP measures of two populations (e.g. set of projects), the statistical

Table 8.2.: Confusion matrix for a binary classifier.

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

difference might be significant, but the effect size could be negligible and therefore the difference has no practical relevance. The metric is nonparametric and thus makes no assumptions about sample sizes, shape, or spread of the compared distributions. It is used in combination with hypothesis testing, which may indicate statistical significance on a metric, but the amount of that discrepancy remains unknown. Cliff's delta ranges from 0 ... 1.0 and guidelines suggest the following interpretation (Grissom and Kim 2012). A value of $|\delta| \leq 0.147$ is considered *negligible*, $0.147 < |\delta| \leq 0.33$ is *small*, $0.33 < |\delta| \leq 0.474$ is *medium*, and otherwise the effect size is *large*.

8.3.5. Precision, Recall, and F-score

The performance of a binary classifier such as the TLSA classifier can be judged using *precision* and *recall*, inherently calculated for the *positive* class. These metrics are defined as

$$\begin{aligned} \text{Precision}_{\text{pos}} &= \frac{TP}{TP + FP} \\ \text{Recall}_{\text{pos}} &= \frac{TP}{TP + FN} \end{aligned} \tag{8.4}$$

TP is the number of *true positives*, i.e. the classifier correctly identified the positive case. The number of *false positives* is written as FP, where the classifier incorrectly identified the positive case. The opposites are *false negatives* (FN), i.e. the classifier incorrectly identified the negative case. Lastly, there are *true negatives* (TN) in which the classifier correctly identified the negative case. (Olson and Delen 2008) Precision is the number of correctly identified positive cases, out of all positive identified cases. On the other hand, recall specifies the number of correctly identified positive cases within all existing positive cases. The values stem from the confusion matrix, which records the achieved counts for the classifier. The confusion matrix is special table layout used to visualize these counts (see Tab. 8.2).

Precision and recall also can be calculated for the negative class, i.e. $\text{Precision}_{\text{neg}}$ and $\text{Recall}_{\text{neg}}$, in a similar way as shown in Equation 8.4. The precision and recall for both classes can be calculated by simply averaging the respective values. However, a weighted average is also possible

$$\text{Precision}_{\text{wavg}} = \frac{\text{Precision}_{\text{neg}} \cdot (TN + FP) + \text{Precision}_{\text{pos}} \cdot (FN + TP)}{TN + FN + FP + TP}$$
$$\text{Recall}_{\text{wavg}} = \frac{\text{Recall}_{\text{neg}} \cdot (TN + FP) + \text{Recall}_{\text{pos}} \cdot (FN + TP)}{TN + FN + FP + TP},$$

which takes the actual class distribution into account, i.e. the number of samples in the negative class ($TN + FP$), and those of the positive class: $FN + TP$. This is especially suitable in case there is an imbalance in the existing positive and negative examples.

The F_β-score combines precision and recall into a single measurement to assess the accuracy of a classification approach

$$F_\beta = (1 + \beta)^2 \cdot \frac{\text{Precision}_{\text{pos}} \cdot \text{Recall}_{\text{pos}}}{(\beta^2 \cdot \text{Precision}_{\text{pos}}) + \text{Recall}_{\text{pos}}}$$

Setting $\beta = 1$ results in the geometric mean of precision and recall and gives equal importance for both measures. Other common values are $\beta = 2$, which favors recall, or $\beta = 0.5$ which weighs precision over recall.

Table 8.3.: Studied parameter settings of TraceScore.

Configuration name	Issue-to-issue trace links	N_{bug}	N_{req}	$D_{\text{bug}}[days]$	$D_{\text{req}}[days]$
Baseline	✓	10	20	365	365
No requirements	✓	10	0	365	365
No explicit dependencies	-	10	20	365	365
All source code files	✓	∞	∞	365	365

8.4. Experiment I: Effectiveness of Similar Issue Component TraceScore (RQ-1)

Experiment I evaluates the effectiveness of the similar issue component TraceScore. A baseline configuration for the four parameters of TraceScore is shown in Table 8.3. These values were determined by investigating the average number of fixed source code files in the dataset (see Tab. 8.1), which is 7.2 for bug reports, and 18 for requirements across all projects. Additionally, bugs and requirements older than $D_{\text{bug}} = D_{\text{req}} = 356$ days are not considered when calculating the score for the current bug report. This configuration is chosen as an initial choice when looking at the projects' lifetimes. The quality of this choice is evaluated by also investigating other configurations: 90 days (quarter of baseline history), 180 days (half of baseline history), and 730 days (double baseline history) for both values, D_{bug}, and D_{req} (see Figures A.1, A.2).

All project data from dataset GS is used for evaluation (see schema in Fig. 8.2). The achieved metrics in terms of Top@k, MAP, and MRR are captured and compared among the evaluated algorithms (see Fig. 8.2). Figure 8.3 visualizes the captured MAP and MRR metrics, and detailed results are provided in Table A.1 in the appendix. The x-axis represents the projects, and the y-axis the achieved MAP and MRR values, respectively. These values are represented as color coded bars for the studied algorithms, whereas a higher bar represents a better performance.

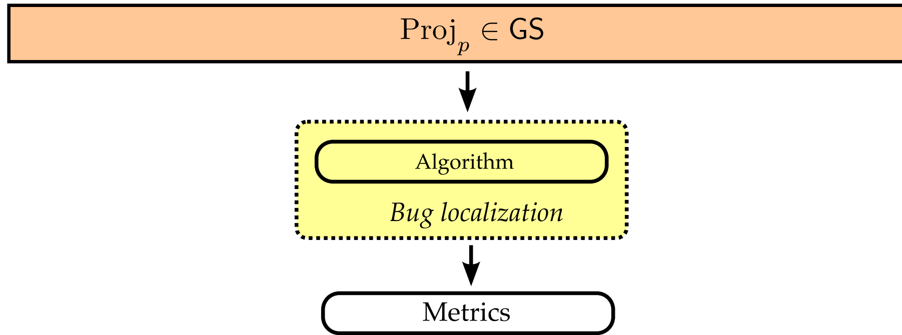
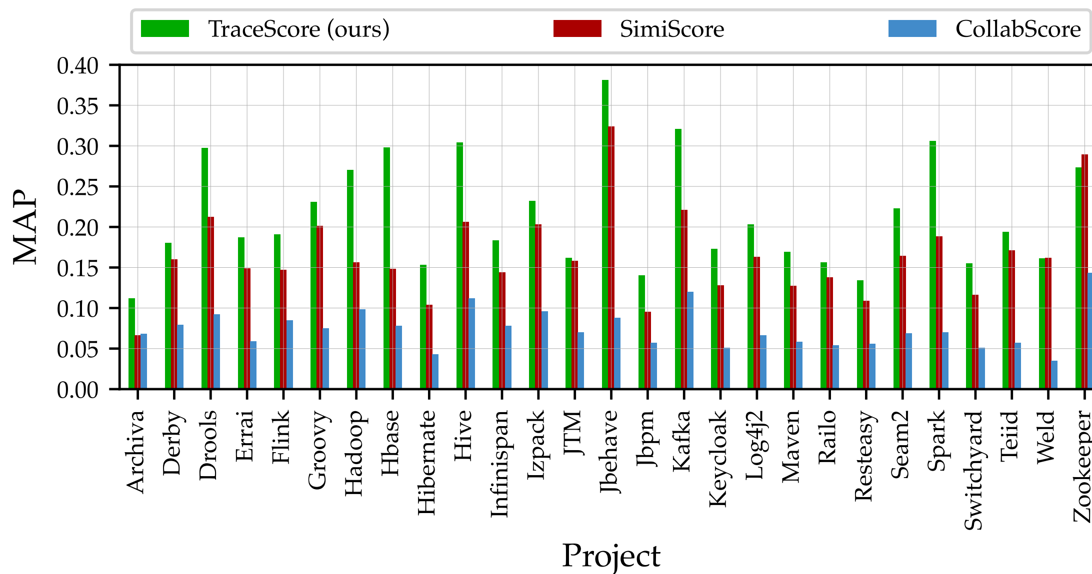


Figure 8.2.: Visualization of experiment I to run the algorithms TraceScore, SimiScore, and CollabScore on a project $Proj_p$.

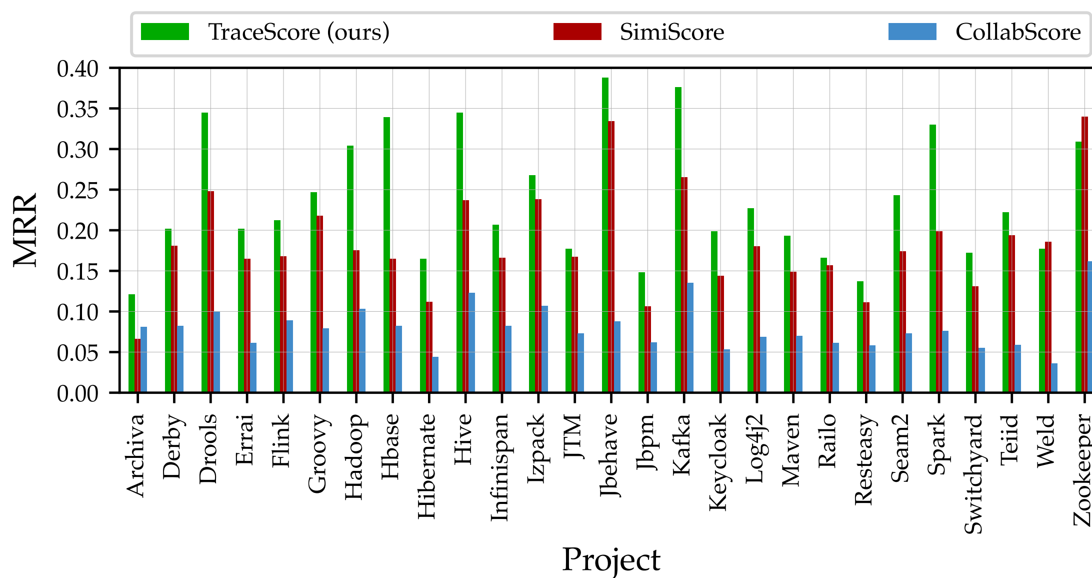
8.5. Experiment II: Impact of TraceScore Parameterization (RQ-2)

Experiment II evaluates the influence of TraceScore parameters, three configurations derived from baseline configuration were created (see Tab. 8.3).

The first configuration “No requirements” only considers previously resolved bug reports and no requirements. It therefore resembles the behavior of SimiScore in this aspect. Thus, all requirement information that modified source code files (see Tab. 8.1) and that might also introduce bugs is ignored. The second configuration “No explicit dependencies” ignores explicit trace links among issue artifacts in the ITS and only relies on text similarity. In this configuration TraceScore is not allowed to leverage existing issue-to-issue trace links captured during dataset mining. The last configuration “All source code files” applies no filtering based on the amount of modified source code files. The schema to perform this experiment in shown in Figure 8.4. The achieved performance values in terms of MAP and MRR compared to baseline configuration of TraceScore are shown in Figure 8.5.



(a) MAP



(b) MRR

Figure 8.3.: Comparision of CollabFilter, SimiScore, and TraceScore in terms of MAP and MRR (higher is better).

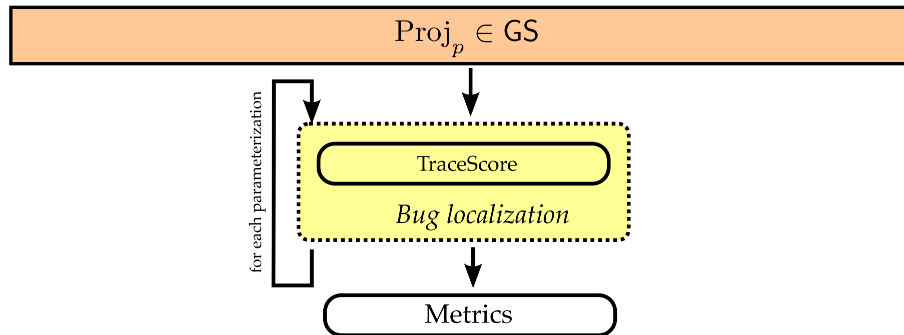


Figure 8.4.: Visualization of experiment II to evaluate different parameterization of TraceScore on Proj_p.

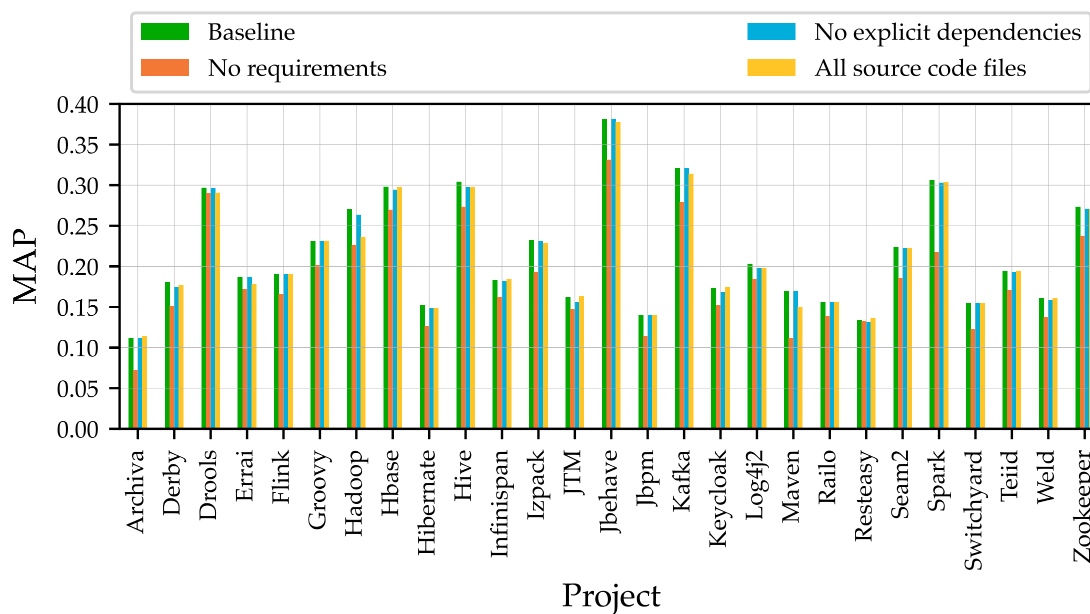
8.6. Experiment III: Effectiveness of IR-Based Bug Localization Algorithm using TraceScore

TraceScore is a similar issue component for a bug localization algorithm. ABLoTS (see Sec. 6.5) was proposed as a IR-based bug localization algorithm using this component. In experiment III, the bug localization performance in terms of Top@k, MAP, and MRR is compared with IR-based bug localization algorithms BLUiR, AmaLgam, as well as the improved version of BLUiR: LuceneScore.

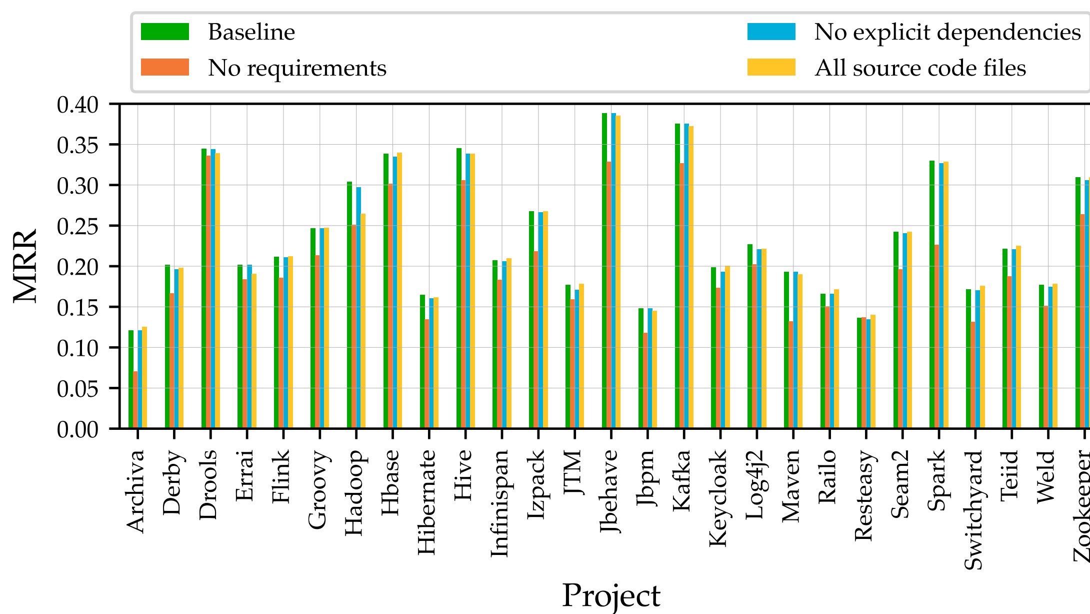
8.6.1. Training and Running ABLoTS on a Project

ABLoTS leverages machine learning techniques to predict a mapping from features of the bug report to source code files pairs to decide whether the respective file needs to be fixed. Thus, no predefined calculation scheme, such as the one used by BLUiR or AmaLgam (see Sec. 6.2), is used to programmatically calculate a score used to rank the source code files. The used classifier algorithm is a random forest, which is trained individually for each project $\text{Proj}_p \in \mathsf{GS}$ as follows.

First the feature matrix $\mathbf{X}$ and the target vector $\mathbf{y}$ is calculated as described in Section 6.5. Next, $\mathbf{X}$ and $\mathbf{y}$ are split into a training and testing set. The split is time based by chronologically sorting the bug reports according to their resolved date and using approximately the first 80% for training (*training bug reports*), and the remaining 20% for testing (*testing bugs reports*). $t_{p,80\%}$ marks this split point. A time based split procedure is necessary, because calculating TraceScore requires the notion of "previously fixed bug reports", which may not hold by randomly selecting bug reports for testing. Once the bug reports used for training and testing are determined, $\mathbf{X}$ and



(a) MAP



(b) MRR

Figure 8.5.: Comparison of different TraceScore configurations (see Tab. 8.3) in terms of MAP and MRR (higher is better).

$\mathbf{y}$ are split in $\mathbf{X_{train}}, \mathbf{X_{test}}$ and $\mathbf{y_{train}}, \mathbf{y_{test}}$ accordingly, i.e. whether the respective feature vector belongs to a bug report from the training or testing bug reports. Afterwards the random forest classifier is trained using $\mathbf{X_{train}}$ and $\mathbf{y_{train}}$.

Special care needs to be taken for the training, because feature matrix $\mathbf{X}$ is highly imbalanced and contains much more feature vectors $\mathbf{x}^{(i)}$ with $y^{(i)} = 0$. These instances describe bug report to source code file pairs, where the source code file doesn't need to be modified. Therefore a classifier could simply predict $y^{(i)} = 0$ for any given feature vector $\mathbf{x}^{(i)}$, resulting in no predictive power overall. To address this issue, $\mathbf{X}$ is subsampled to the minority class, i.e. instance representing that the source code file is necessary to be modified to resolve the bug report. The subsampling is done on a per bug report basis (see Tab. 6.1). All feature vectors for a bug report representing fixed source code files are taken, and a random *equal* amount of feature vectors for source code files that weren't modified. Training the classifier and evaluation is repeated five times to mitigate the random effects of subsampling. The reported metrics Top@k, MAP, and MRR on the testing bug reports is represented as macro averaged values of the individual ABLoTS evaluations.

A random forest has a set of *hyperparameters*, i.e. parameters that are not determined by classifier training and thus require manual setup. For example, this includes the number of trees in the forest or their individual depth. The best hyperparameter settings are dependent on $\mathbf{X}$, $\mathbf{y}$ and are usually unknown. Therefore different hyperparameter values are considered to find the best setup. The identification of this setup is done by evaluating the performance (e.g. in terms of MAP) on data different from the training and testing data. However, introducing the so called *validation data* for every project would drastically reduce available training data which is used to learn the model. For example using 60% of bug reports for training, 20% for validation, and the remaining 20% testing. Instead, a k-fold cross-validation approach with $k = 5$ was chosen for ABLoTS, which solely operates on the 80% training data *without* touching the 20% testing data. The execution strategy is shown in Figure 8.6. The subsampled $\mathbf{X_{train}}$ and $\mathbf{y_{train}}$ data is split in five equal parts (the folds) on a per bug report basis, such that each fold contains the same amount of bug reports. Using k-fold cross-validation means, that the classifier is trained on $k-1$ folds, combined into one training set, and then the last fold is used as test set (Albon 2018). This is repeated k times, each time using a different fold as test set. The achieved performance of the model for each of the k iterations is then averaged to produce an overall measurement. In case of ABLoTS, the first iteration to train the classifier uses the folds $1, 2, 3, 4$, and is tested using fold 0, and the achieved metrics are saved. For the next iteration, the folds $0, 2, 3, 4$ are used for training, and fold 1 for testing. The 5-fold cross-validation is executed for each hyperparameter combination and the one yielding the highest MAP value is selected as final setting. Eventually, this hyperparameter configuration is used to train the classifier, which is than applied to $\mathbf{X_{test}}, \mathbf{y_{test}}$.

Figure 8.6.: Visualization of k-fold cross-validation with $k = 5$ on training data (adopted from (Sklearn Documentation 2021)). Five iterations with different combinations of training and testing folds are performed, the achieved metrics are collected and finally averaged. For each iteration different portions from the training data is used for actual training, and one portion from the training data is used for validation, i.e. testing the hyperparameter performance. The actual testing data is not considered for cross-validattion.

To recap, running ABLoTS on a $\text{Proj}_p \in \mathsf{GS}$ is (a) subsampling the training bug reports, (b) applying a 5-fold cross-validation (i.e. not wasting training data) on this data for every hyperparameter combination, (c) selecting the hyperparameters achieving the largest MAP value, (d) individually train five classifiers on subsampled training data using the best hyperparameter settings, run the classifiers on testing data, and lastly (f) macro average the individual metrics of each test run (see Fig. 8.7).

The other three bug localization algorithms require no training and are simply applied on the test set of bug reports (see Fig. 8.8) and performance measures are captured. For BLUiR and LuceneScore, the project's source code is indexed at the point in time, when the last training bug report was resolved. The composer component of AmaLgam has two parameters a and b. These values were empirically determined by the authors of AmaLgam and set to $a = 0.2$ and $b = 0.3$, which are therefore used for all setups in the evaluation. The chosen values prioritize the source code structure component over the similar issue component, and both over the history component.

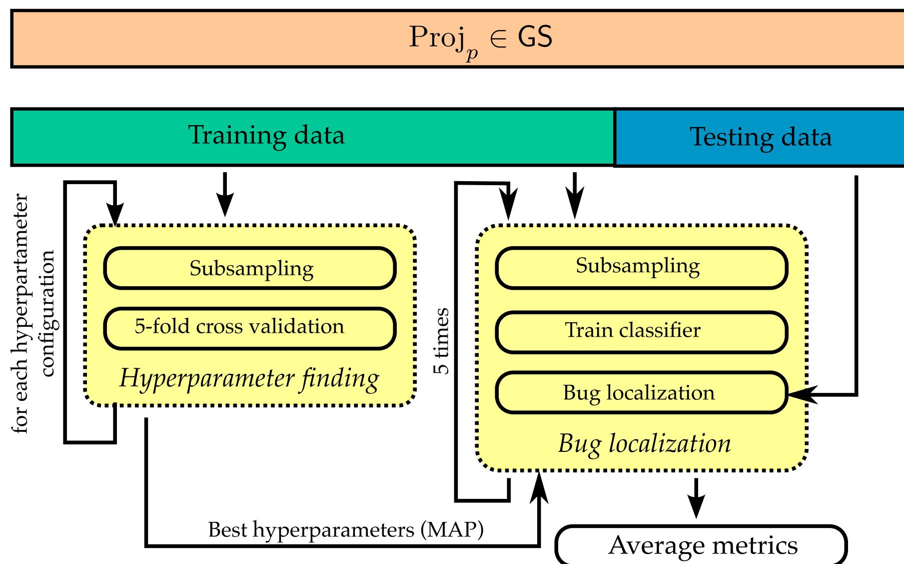


Figure 8.7.: Visualization of experiment III to apply ABLoTS to a project. First, the project data is split into training and test data. Next the training data is used to find best hyperparameters. Using these, the classifier of ABLoTS is trained and applied to the testing bug reports.

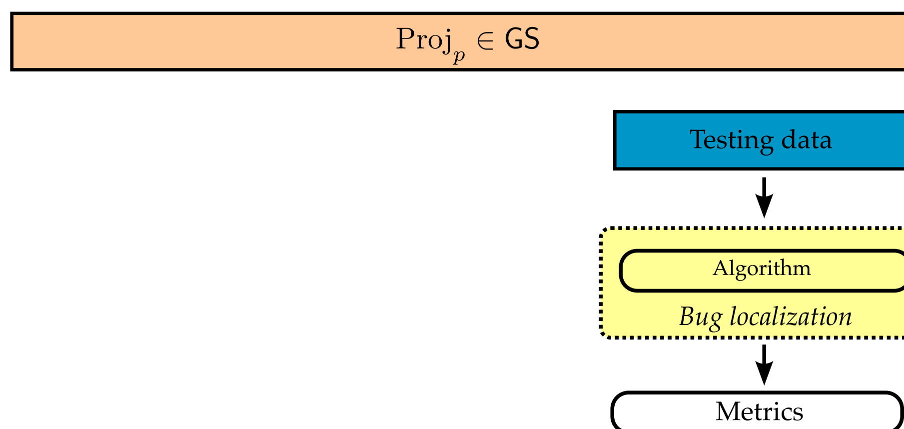


Figure 8.8.: Visualization of experiment III for bug localization algorithms BLUiR, AmaLgam, and LuceneScore on Proj$_p$.

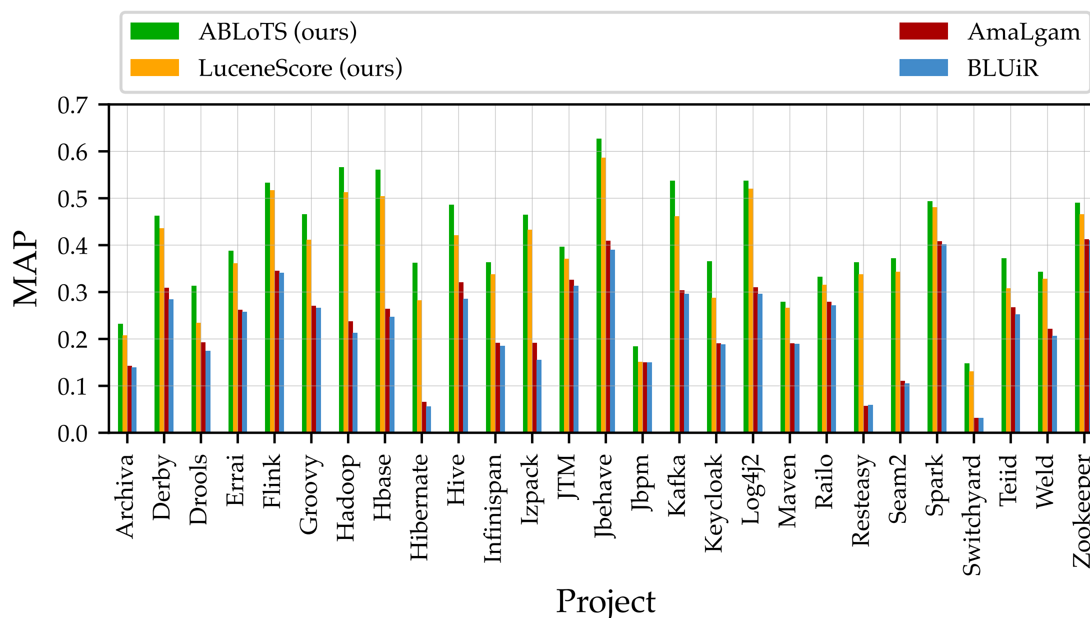


Figure 8.9.: Comparison of different bug localization algorithms in terms of MAP on the testing data of each project (higher is better).

8.6.2. Results

The achieved performance in terms of MAP for the four algorithms on the testing data is shown in Figure 8.9. It exhibits, that the proposed algorithms ABLoTS and LuceneScore outperform the previous work of BLUiR and AmaLgam. Detailed metrics also including Top@k, and MRR are listed in Table A.2.

8.7. Experiment IV: Effectiveness of Trace Link Set Augmentation (RQ-4)

Experiment IV evaluates the proposed TSLA algorithm (see Sec. 7.4). It serves as a pre-requisite to investigate IR-based bug localization performance on projects containing augmented trace links, which is evaluated in *experiment V*.

8.7.1. Training and Running the Trace Link Set Augmentation Classifier

Using the same training and test split introduced in *experiment III* for training and running TSLA is not possible. This would result in augmented issue-to-commit trace links only on a portion of all project data, i.e. no augmented trace links are available in the training part of the dataset. Augmenting these links using the trained TLSA classifier would violate a fundamental principle in machine learning to never mix training and testing data. However, the trace link set in training data also needs to be augmented, so that the bug localization algorithms, foremost ABLoTS, could benefit from these links.

Therefore a more sophisticated scheme is used to augment issue-to-commit links throughout the *whole* lifetime of a project. This is accomplished by training a *project independent* classifier, which then augments trace links in a different project. For a given project $p \in \mathsf{GS}_{\text{project-names}}$ and one of its reduced issue-to-commit link sets $\text{Proj}_{p,\text{red},i}$, the following *leave one out* scheme is used. A classifier is trained on all projects, the *training projects* different from p, i.e. $\mathsf{GS} \setminus \text{Proj}_p$, and then augments trace links in $\text{Proj}_{p,\text{red},i}$. The feature matrices $\mathbf{X}$ and target vectors $\mathbf{y}$ for the training projects are calculated using GS and combined (stacked) into on large matrix $\mathbf{X}_{\mathbf{train}}$ and vector $\mathbf{y}_{\mathbf{train}}$, respectively. Next, $\mathbf{X}_{\mathbf{test}}$ and $\mathbf{y}_{\mathbf{test}}$ for the project to augment are created from the specific reduced data $\text{Proj}_{p,\text{red},i}$. Now, the classifier is trained on $\mathbf{X}_{\mathbf{train}}, \mathbf{y}_{\mathbf{train}}$, and applied on $\mathbf{X}_{\mathbf{test}}$, which yield the augmented project $\text{Proj}_{p,\text{aug},i}$.

The actual classifier training is similar to the training of ABLoTS described in Section 8.6.1 and also requires finding hyperparameters, and use subsampling. The hyperparameters for the random forest are determined using a 5-fold cross-validation on $\mathbf{X}_{\mathbf{train}}, \mathbf{y}_{\mathbf{train}}$ to retrieve the configuration with the best F_1 score. This metric is chosen as tradeoff between precision (the augmentation algorithm should only create *correct* trace links), and recall (the augmentation algorithm should re-create *all* trace links). $\mathbf{X}_{\mathbf{train}}$ is highly imbalanced, i.e. there are much more instances representing an issue-commit pair without a link, than those representing a linked pair. Thus, $\mathbf{X}_{\mathbf{train}}$ and $\mathbf{X}_{\mathbf{test}}$ are randomly subsampled to the minority class to balance the data

as explained in training the ABLoTS classifier. To mitigate the random subsampling effects, five different random forests using the best hyperparameter settings are trained, and are individually applied to $\mathbf{X_{test}}$. This possibly results in different predictions, i.e. one classifier suggests a trace link between a particual issue-commit pair, and another random forest recommends not to trace these two artifacts. To break these ties, a voting with simple majority is used, i.e. three out of five classifiers need to recommend the issue-commit pair to be considered as a correct trace link. The recommended trace links are added to $\text{Proj}_{p,\text{red},i}$ and thus yield $\text{Proj}_{p,\text{aug},i}$. Figure 8.10 visualizes this execution scheme. Eventually, the augmentation routine is performed for each project, repeated five times once for every reduced link dataset.

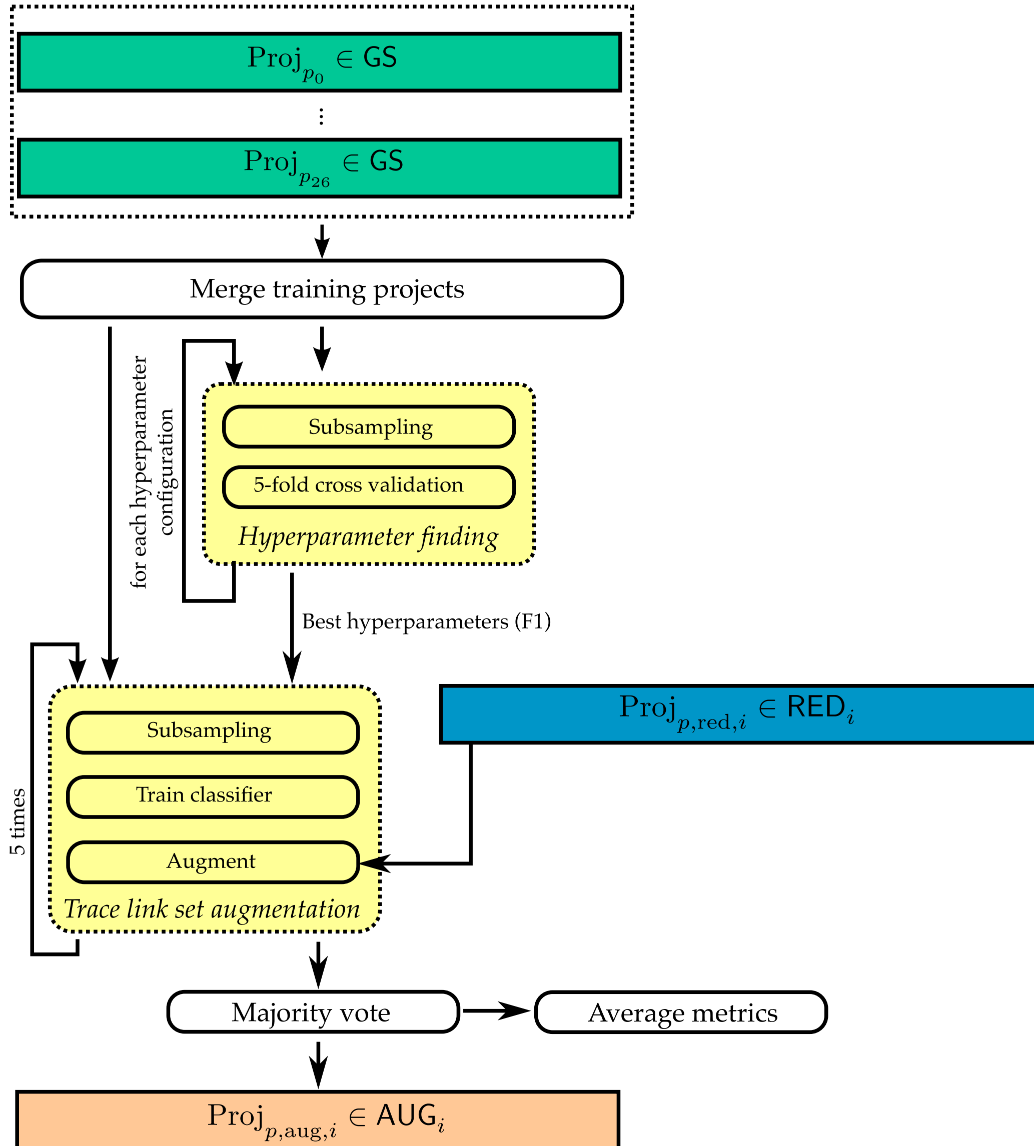


Figure 8.10.: Visualization of experiment IV. All project data, except for the project p to augment, are used to train the TSLA classifier. This project independent classifier is used to augment one reduced link dataset $Proj_{p,red,i}$ yielding the respective augmented project $Proj_{p,aug,i}$. This routine is repeated for all five reduced linked project data for every project.

8.7.2. Results

The augmentation performance is measured in terms of precision, and recall and is shown in Figure 8.11. To get single measures, the individual results achieved on each $\mathrm{Proj}_{p,\mathrm{aug},i}$ per project are averaged. There are two color coded bars per project on the x-axis. The blue bar represents the achieved precision, and the green one the recall. Both metrics range from 0% to 100% (higher values are better).

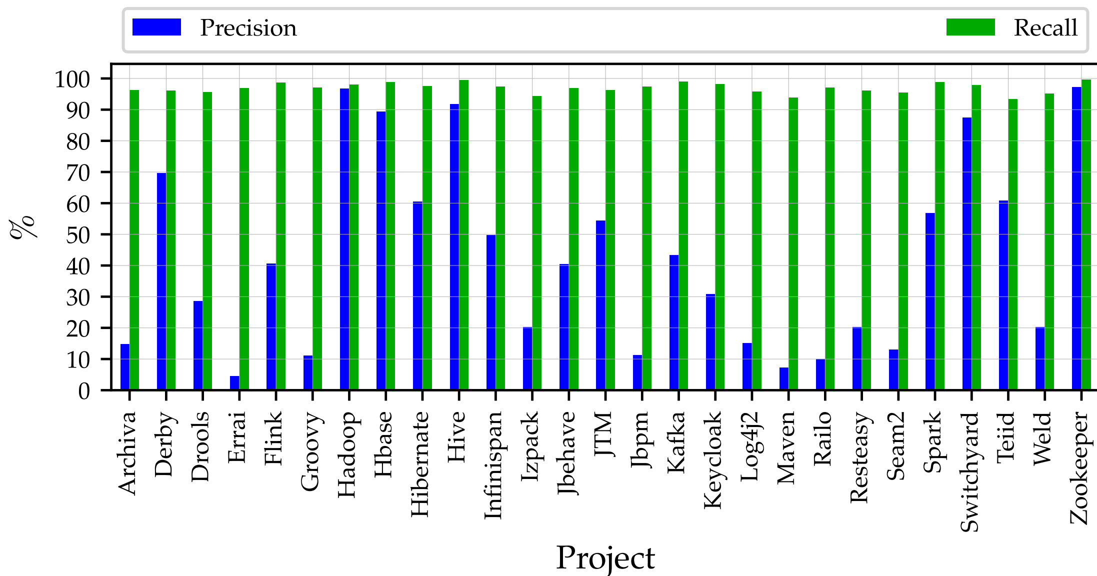


Figure 8.11.: Trace link set augmentation performance in terms of precision, recall, for all projects (higher is better). 40% of the original trace links per project were randomly removed and TSLA algorithm was trained to restore them. This was repeated five times, and the depicted measures are averages (micro).

8.8. Experiment V: Effectiveness of IR-Based Bug Localization Algorithms on Projects with Augmented Trace Link Sets (RQ-5)

Experiment V is similar to *experiment III* (see Sec. 8.6). Instead of performing bug localization on GS, the five issue-to-commit trace link augmented datasets AUG_i are used (see Fig. 8.12). Each $\mathrm{Proj}_{p,\mathrm{aug},i} \in \mathsf{AUG}_i$ is split into training bug reports and testing bug reports using the identical split points $t_{p,80\%}$ as described in *experiment III* (Sec. 8.6). The algorithms LuceneScore and BLUiR do not leverage trace link information and thus are unaffected by the augmentation process. BLUiR, AmaLgam and LuceneScore require no training and are simply applied to the testing data of the augmented trace links sets (see Fig. 8.13). ABLoTS is trained on training bug reports, and applied to testing bug reports.

The achieved metrics Top@k, MAP, and MRR for each augmented project $Proj_{p,\mathrm{aug},i}$ are collected and afterwards averaged. Fig. 8.14 shows the performance in terms of MAP. For ease of comparision, the results from experiment III for ABLoTS, LuceneScore, AmaLgam, and BLUiR are also depicted. The algorithms labeled “augmented” differentiate the performance on GS and the averaged results on AUG_i.

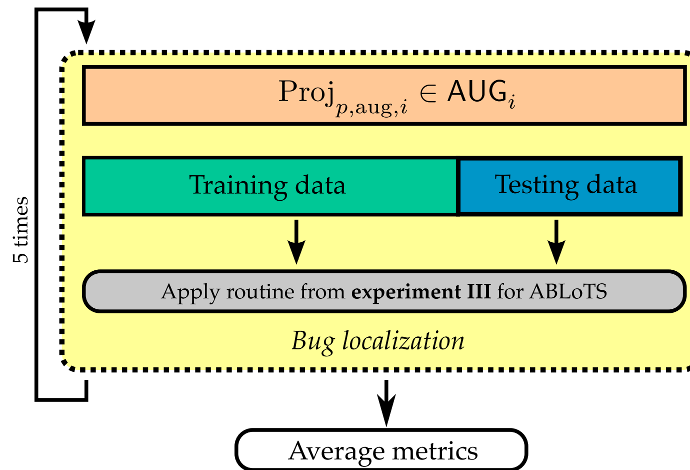


Figure 8.12.: Visualization of experiment V to apply ABLoTS to a project, which is similar to experiment III (see Figure 8.7). Bug localization is performed on each augmented project $\text{Proj}_{p,\text{aug},i}$ and the achieved results are averaged. This procedure is repeated for all projects.

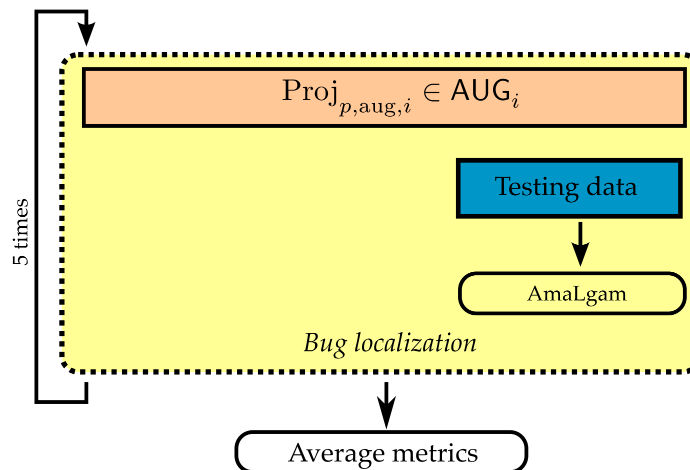


Figure 8.13.: Visualization of experiment V for AmaLgam for each project $\text{Proj}_{p,\text{aug},i}$. Bug localization is performed on each augmented project $\text{Proj}_{p,\text{aug},i}$ and the achieved results are averaged. This procedure is repeated for all projects.

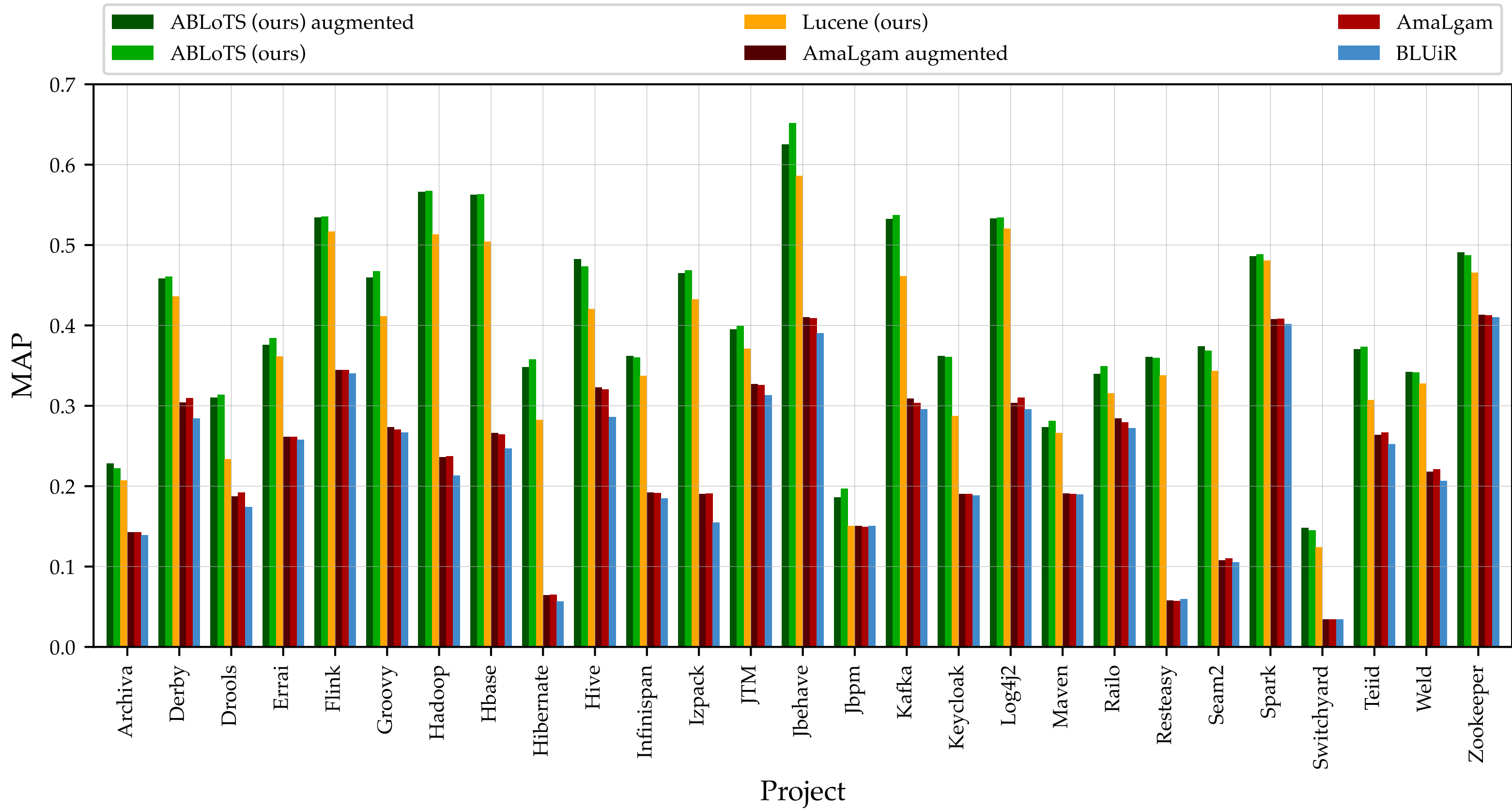


Figure 8.14.: Comparision of different bug localization algorithms in terms of MAP (higher is better). The bug localization is performed on AUG_i and averaged for the algorithms ABLoTS (ours) and AmaLgam. For ease of comparision, the performance in terms of MAP from experiment II are depicted as well. The algorithms labeled with "augmented" were applied on augmented datasets, the others on reference GS.

9. Discussion

In this section the results from the five evaluation experiments are discussed. Each experiment is handled separately and the corresponding research question is answered. The section ends with an analysis of potential threats to validity and how they were addressed.

9.1. RQ-1 - Effectiveness of Similar Issue Component TraceScore

TraceScore with baseline configuration outperforms previous approaches, i.e. CollabScore and SimiScore, in 25 out of 27 projects (see Figs. 8.3a, 8.3b). SimiScore achieves slightly higher Top@1 values for projects Weld and Zookeeper, and thus resulting in higher values in terms of MAP and MRR. CollabScore performs worst in all metrics and on every project. This most likely stems from its rather simple algorithm only relying on textual similarity of bug reports' summaries (see Sec. 6.2.2). On average, TraceScore achieves a 53% better Top@1, 24% better Top@5, 23% better Top@10, 33% better MAP, and 32% better MRR than SimiScore. The nonparametric[1] Wilcoxon signed-rank test (Wilcoxon 1992) showed, that the null hypothesis stating *"TraceScore is equal or worse than SimiScore"* is rejected for every metric with a significance level of $p < 0.05$. The effect size on the Top@1 metrics is considered large with $|\delta| = 0.5$, and medium ($|\delta| \approx 0.4$) for the other metrics.

The following example illustrates the ability of TraceScore to utilize additional artifact and trace link data. Four source code files were modified to fix bug report `DERBY-4214`[2] from project Derby. Both, SimiScore and TraceScore rank one of the modified source code files, `DD_Version.java`, on 4$^{\text{th}}$ place. However, TraceScore also correctly ranks `DataDictionaryImpl.java` on 3$^{\text{rd}}$ place. This source code file was previously modified to implement improvement `DERBY-3769`[3].

1. A nonparametric test does not assume a specific distribution, e.g. normal distribution, of the independent variables.
2. DERBY-4214: https://issues.apache.org/jira/browse/DERBY-4214, fixed files: https://bit.ly/3jf6lbg
3. DERBY-3769: https://issues.apache.org/jira/browse/DERBY-3769, fixed files: https://bit.ly/3gX5kCQ

Finding TraceScore effectively localizes bugs in source code files. It outperforms existing similar issue components in terms of Top-1, Top-5, Top-10, MAP, and MRR. In particular, it achieves on average a 53% higher Top-1, and 33% better MAP than its closest competitor resulting in more correctly identified source code files on higher ranks.

9.2. RQ-2 - Impact of TraceScore Parameterization

Changing the parameterization of TraceScore, i.e. N_{bug}, N_{req}, D_{bug}, D_{req}, affects its performance. The chosen temporal values $D_{\text{bug}} = D_{\text{req}} \equiv 365$ for TraceScore baseline yield reasonable results in terms of MAP and MRR compared to other choices (see Figs. A.1, A.2). Especially shortening the history length to 180 days, or even down to 90 days, significantly reduces the performance in most projects. Using these parameter values, TraceScore loses information by limiting the time interval of previously resolved bug reports and implemented requirements. For example, the bug report `DERBY-6705`[4] was created and resolved in August 2014. This report has an explicit trace link to bug report `DERBY-6375`[5], which was resolved in October 2013. However, this information is lost in case the history is too short.

Increasing the considered history by setting D_{bug} and D_{req} to 730 days provides slightly better results as the chosen baseline value of 365 days. Thus, it is possible the extract additional information from even older bug fixes and implemented requirements. However, this process does not continue indefinitely. In case of the unbounded value choice for D_{bug} and D_{req}, i.e. to use all available past artifacts, MAP and MRR generally decrease below that achieved for 730 days. Sometimes it is even worse than baseline, e.g. for project Hbase. This indicates, that very old source code changes provide no information for a current bug report at hand. There is one exception, project Zookeeper, where utilizing all past issues results in highest MAP and MRR.

Despite changing TraceScore's history length, modifying the other parameters also affects the performance of TraceScore in terms of MAP and MRR as shown in Figures 8.5a, 8.5b. The most impact on TraceScore performance in this regard has the exclusion of requirement artifacts (see "No requirements" configuration). Thus, all previous source code modifications and not only bug fixes provide useful information for handling the current bug report. The achieved MAP and MRR metrics drop for all projects when requirements are excluded. The highest drop occurs for project Archiva, where MAP is reduced by 35% and MRR by 41%. Ignoring

4. https://issues.apache.org/jira/browse/DERBY-6705
5. https://issues.apache.org/jira/browse/DERBY-6375

requirement artifacts for bug localization has nearly no effect on project Drools. Nevertheless, even without using requirement artifacts, TraceScore achieves higher MAP values than SimiScore in 19 projects, and better MRR values in 17 projects.

The second configuration, "No explicit dependencies", uses the same parameters as baseline configuration, but ignores explicitly defined trace links among issue artifacts in the issue tracking system. Interestingly, this does not affect the performance of TraceScore, neither in terms of MAP nor MRR. Considering only bug localization, the developers could skip the elaborate task to create these trace links in the first place. An explanation might be, that linked artifacts already have a high textual similarity, and thus a high edge weight in the constructed traceability graph.

The last configuration, "All source code files", does not filter issue artifacts based on the number of modified source code files. This results in nearly identical metrics as the baseline configuration. However, it has an effect on the cost to compute TraceScore, because potentially a large amount of ratios need to be summed up (see Eq. 6.6). Thus, a simpler solution, i.e. baseline configuration, should be preferred.

The parameter study underlines the chosen TraceScore baseline configuration is a sensible choice for studied projects.

Finding Requirement artifacts in the issue tracker improve the bug localization performance of TraceScore. Explicitly created issue-to-issue trace links only provide a minor impact on the performance and thus their creation should not be the focus of the developers. The baseline parameterization of TraceScore performs best and can be used for all projects.

9.3. RQ-3 - Effectiveness of an IR-based Bug Localization Algorithm using TraceScore

The competitor algorithms BLUiR and AmaLgam both perform worse in terms of MAP compared to the novel algorithms LuceneScore, and ABLoTS (see Fig. 8.9). On average, ABLoTS achieves more than 100% of Top@5, Top@10, MAP, and MRR compared to AmaLgam, and even 150% at Top@1. The measures for AmaLgam are comparable to those reported by Wang et al. (Wang and Lo 2014) on their dataset consisting of only four projects. AmaLgam also consistently outperforms BLUiR as in evaluation in that paper, but the difference in terms of MAP is much closer on the dataset used in this thesis.

LuceneScore also performs very good and ranks second on each project in terms of MAP. For many bug reports the algorithm is able to identify one out of the on average five (see Tab. 8.1) modified source code files per bug report and ranks it first.

Bug reports often directly mention class names or identifiers found in the source code. For example, bug report `ZOOKEEPER-2786`[6] from project Zookeeper reads *"Flaky test: org.apache.zookeeper.test.ClientTest testNonExistingOpCode"*, which contains a full qualified name of a test case implemented in Java class `ClientTest`. The modified source code files for that bug report are `ClientTest.java`, `NettyServerCnxn.java`, and `TestableZookeeper.java`. LuceneScore ranks `ClientTest.java` first, because matches are found within identifiers and class names and thus $\text{lucene}_{\text{comb}}$ (`ClientTest.java, ZOOKEEPER-2768`) is high (see Sec. 6.4). It is the common practice in the Java programming language to implement a single class in a source code file and name that file after the class. Thus, `ClientTest.java` represents the implementation of class `ClientTest`, which is mentioned in the bug report and is correctly identified by LuceneScore. Neither `NettyServerCnxn.java` nor `TestableZookeeper.java` appear in the ranked source code file list resulting in a Top@1 hit and thus a perfect average precision for this bug report. A similar example is bug report `KAFKA-4840`[7] of project Kafka whose description starts with *"There are several problems dealing with errors in BufferPool.allocate,(int size, long maxTimeToBlockMs) [...]"*. In this case a whole method signature is contained in the bug report. The source code files modified for the bug report are `BufferPool.java`, and `BufferPoolTest.java`. Both files are on high places of the ranked file list created by LuceneScore, because of high score values from $\text{lucene}_{\text{comp}}$.

However, if there are no direct clues mentioned in the bug report, i.e. having a generic summary and description, LuceneScore performs weak. For example, bug report `KAFKA-6210`[8] reads *"IllegalArgumentException if 1.0.0 is used for inter.broker,.protocol.version or log.message.format.version"* with the fixed source code file `ApiVersionsResponseTest.java`. Misled by the description, LuceneScore ranks arbitrary source code files on top, because of the terms "IllegalArgumentException" or "log.message", which often appear in the source code.

BLUiR and LuceneScore, share the same fundamental concepts and thus should produce similar results (see Sec. 6.4). This is true for the above mentioned bug reports, `ZOOKEEPER-2786`, `KAFKA-4840`, for which both algorithms are able to yield a perfect result, and `KAFKA-6210`, for which both fail to identify any source code file that need to be fixed. However, LuceneScore still performs better, because it more often predicts correct source code files at higher ranks as BLUiR. For example, bug report `ZOOKEEPER-2141`[9] reads *"ACL cache in DataTree never removes entries"*, and LuceneScore ranks `DataTree.java` correctly at the first place, whereas BLUiR puts it second and has `Login.java` as top result. Thus the applied modifications,

6. https://issues.apache.org/jira/browse/ZOOKEEPER-2786
7. https://issues.apache.org/jira/browse/KAFKA-4840
8. https://issues.apache.org/jira/browse/KAFKA-6210
9. https://issues.apache.org/jira/browse/ZOOKEEPER-2141

e.g. more sophisticated textual analysis, when implementing LuceneScore proof their usefulness (see end of Sec. 6.4).

ABLoTS achieves higher values on most measures in all projects compared to LuceneScore. An application of the nonparametric Wilcoxon signed-rank test (Wilcoxon 1992) showed, that these differences are significant ($p < 0.05$) and ABLoTS outperforms LuceneScore. Calculating cliff's delta results in $|\delta| \approx 0.2$ for each of the five reported metrics, which indicate small but substantial differences. Thus, combining traceability information from TraceScore with the source code structure information significantly improves bug localization performance.

In order to investigate the contribution of different components TraceScore, SimiScore, and BugCache score on testing data have been calculated and evaluated in terms of MAP (see Fig. 9.1). The achieved MAP values for TraceScore and SimiScore are comparable with those calculated on the whole project data (see Fig. 8.3a), and TraceScore still performs best on most projects. The only exceptions are the projects Jbehave, which only contains ten test bugs and thus is neglectable, project Derby and project Zookeeper, where SimiScore performs best in terms of MAP. The contribution of BugCache is least, ranging from 0 to about 0.1 MAP points per project, whereas the other scores go up to > 0.4 MAP points. Comparing Figure 8.9 and Figure 9.1 shows that the different bug localization algorithm components not only benefit from each other when combined, i.e. the final score is not simply the sum of the component scores. For example, for project Hive, ABLoTS (the combination of LuceneScore, TraceScore, and BugCache) achieves a MAP of 0.48, whereas the individual components achieve 0.42, 0.33, and 0.05 for LuceneScore, TraceScore, and BugCache respectively. This value is much lower as one would expect when considering the individual component scores. One reason for this is, that the different components correctly rank the same candidate source code files on top, resulting in good metrics individually, but not changing the overall metrics when combined. This combination effect is also present for AmaLgam, which combines BLUiR, SimiScore, and BugCache. For project Hive, AmaLgam achieves a MAP of 0.32, and the individual component scores are 0.28, 0.19, and 0.05 for BLUiR, SimiScore, and BugCache.

Finding The novel IR-based bug localization algorithm ABLoTS benefits from TraceScore. It outperforms two competitor algorithms by 100% on all measures. The modifications applied in LuceneScore improved the structured source code search compared to baseline algorithm.

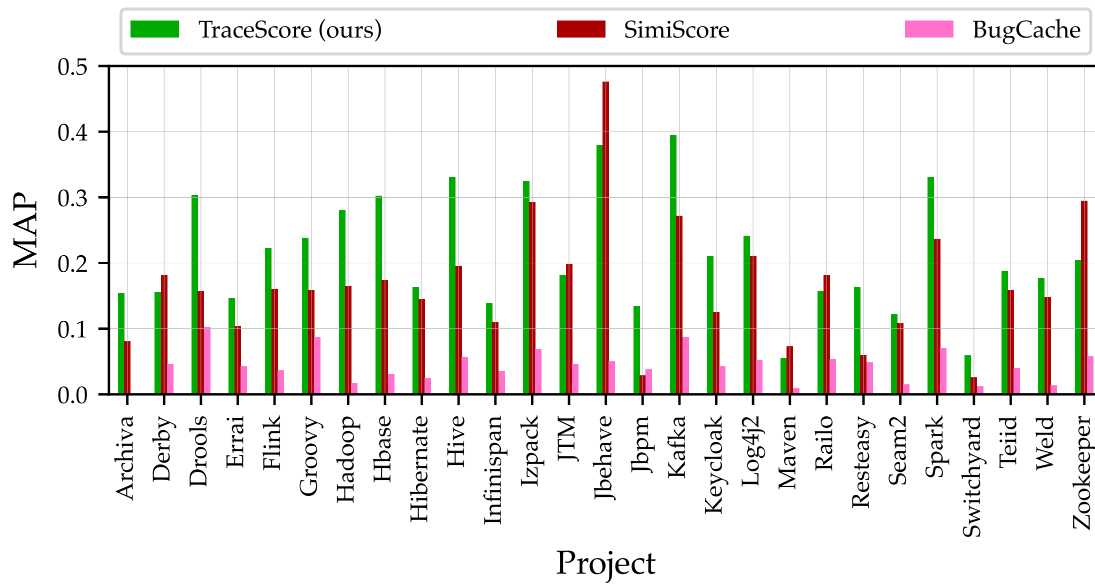


Figure 9.1.: Comparison of different bug localization algorithms components in terms of MAP on testing data of each project (higher is better). TraceScore (ours), and SimiScore are similar issue components used in ABLoTS (ours) and AmaLgam, respectively. The history component BugCache is used in both, ABLoTS (ours) and AmaLgam.

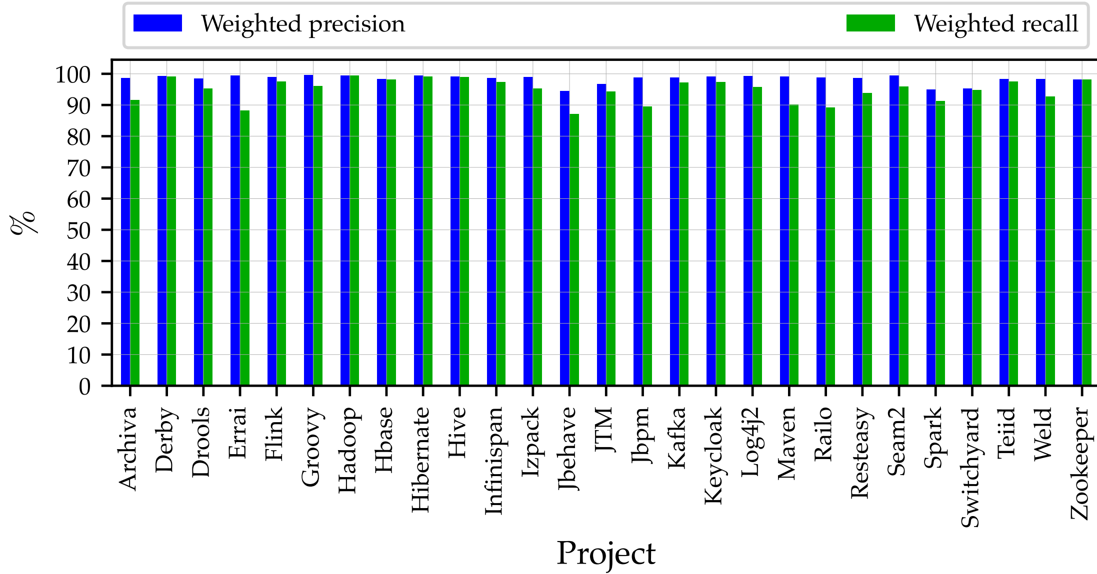


Figure 9.2.: Trace link set augmentation performance in terms of weighted precision and recall for all projects (higher is better). These metrics take the class distribution into account contrasting Figure 8.11. The depicted measures are averages achieved on RED_i.

9.4. RQ-4 - Effectiveness of Trace Link Set Augmentation

The results for augmenting issue-to-commit trace links on the reduced datasets in terms of precision, and recall are shown in Figure 8.11. Overall, the recall for all projects is 90% or higher (97% on average). Thus, the augmentation process creates few false negatives and is able to restore nearly all of the previously removed trace links. The precision of the augmenter is more diverse. Especially for projects Maven, and Railo it is less than 10%, and the lowest precision is achieved for project Errai with 4.5%. Thus, TLSA algorithm creates lots of issue-to-commit trace links, that are not linked (false positives) in gold standard GS. On the other hand, high precision values of 90% or above are achieved for projects Hbase, Hive, and it reaches a maximum of 97% for project Zookeeper, i.e. nearly no superfluous trace links are created. Taking the actual class distribution into account, i.e. the number of nonlinked and linked issue commit pairs, the weighted average precision (see Sec. 8.3) is above 90% for all projects, and 98% on average across all projects (see Fig. 9.2).

The reason for the mixed precision results is likely the construction of the augemented datasets AUG_i (see Eq. 8.3). In gold standard GS only linked issue-commit pairs

exist, in the reduced datasets $\mathsf{GS}_{\text{red},i}$ linked issue-commit pairs are present, as well as nonlinked ones, i.e. artificially removed ones. But, the augmentation process considers all commits from the orginial SEOSS project $\text{Proj}^{(\text{seoss})}$. This also includes commits that never traced to an issue (see column "unlinked" in Tab. 7.1, and issue-commit example pairs in Eq. 7.9). So, the amount of these commits describes the difficulty for the augmentation process. For example, in project $\text{Proj}_{\text{Zookeeper}}$ there are 835 linked issue-commit pairs. In original $\text{Proj}^{(\text{seoss})}_{\text{Zookeeper}}$ are additional 73 nonlinked commits, which matches the low unlinked commit ratio of 8% as reported in Table 7.1. Thus, when creating the features for project Zookeeper, fewer issue-commit combinations are possible that represent a candidate pair which will never be linked. The same applies for other projects with high precision values, e.g. projects Hadoop and Hive. These projects also have a very high issue-to-commit link ratio of 99% and 97%, respectively. Only 267 and 311 unlinked commits are added from $\text{Proj}^{(\text{seoss})}_{\text{Hadoop}}$ and $\text{Proj}^{(\text{seoss})}_{\text{Hive}}$ repectively, constrasting the $\approx 21,000$ and $\approx 9,000$ linked ones.

The TLSA algorithm only achieves 5% precision on project Errai. $\text{Proj}_{\text{Errai}}$ has 542 linked issue-commit pairs. However, the nonlinked commit ratio of 91% in $\text{Proj}^{(\text{seoss33})}_{\text{Errai}}$ results in $\approx 5,500$ possible issue-commit pairs of considered for tracing, which resembles the opposite for projects Hadoop, Hive, and Zookeeper. Every 8th instance in the feature matrices created for $\text{Proj}_{\text{Hadoop,red,0}}$ and $\text{Proj}_{\text{Hive,red,0}}$ encodes a true issue-to-commit pair. For $\text{Proj}_{\text{Errai,red,0}}$ it is every 160th, and for $\text{Proj}_{\text{Maven,red,0}}$ every 130th. Assuming a constant error rate of the augmenter per instance to introduce a false positive this imbalance results in low precision metrics.

9.4.1. Evaluating Additional Trace Link Removal Settings

In datasets RED_i 40% of existing issue-to-commit trace links are removed, emulating developer behavior based on statistics of GS (see Sec. 8.2.2). Other removal rates are also instresting to study the augmentation behavior. For example, does the augmentation perform better in terms of precision, if fewer trace links are removed? Or considering the worst scenario and removing all trace links, and thus answer the question: *"Is it possible to augment a project, which does not have any trace links at all?"*. If so, the time consuming manual trace link creation could be replaced by a fully automated process. The first question is evaluated by repeating experiment IV, but using five datasets $\mathsf{RED}_{i,20\%}$, $i \in \{0, \dots, 4\}$ with only 20% of issue-to-commit trace links removed. The achieved performance in terms of weighted precision and recall is shown in Figure 9.3. Changing the trace link removal rate does not affect the training of the augmentation classifier for a project $p \in \mathsf{GS}_{\text{project-names}}$, because the training process always uses $\bigcup_q \text{Proj}_q$ with $p \neq q$, $q \in \mathsf{GS}_{\text{project-names}}$ (see Fig. 8.10). The only difference is the feature vector generation for $\text{Proj}_{p,\text{red},i,20\%}$, because now

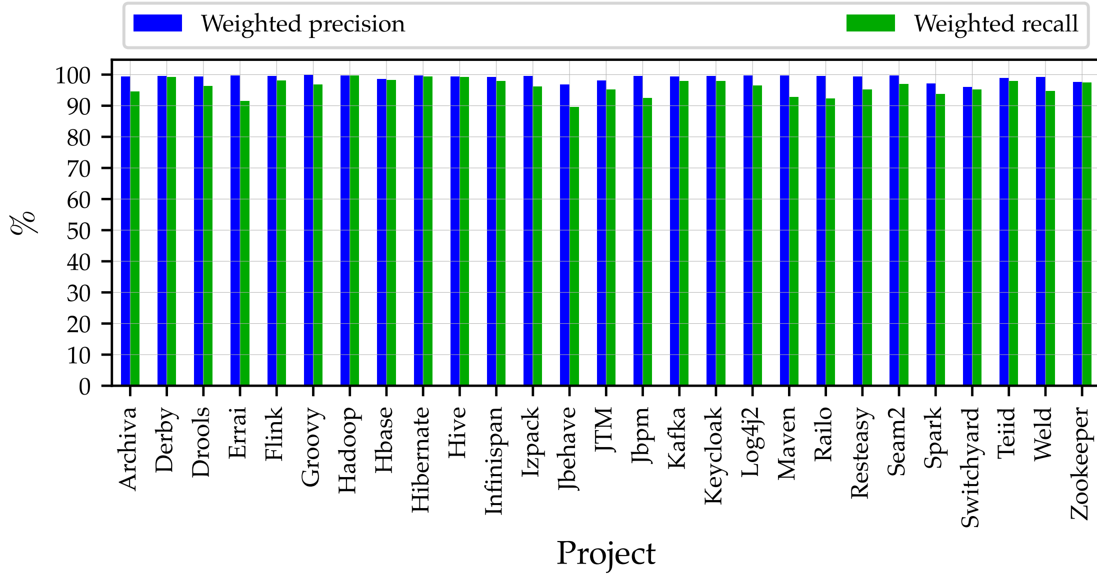


Figure 9.3.: Trace link augmentation performance in terms of weighted precision and recall for all projects (higher is better). Instead of removing 40% of issue-to-commit links, only 20% have been removed. The depicted measures are averages achieved on $\mathsf{RED}_{i,20\%}$.

the feature matrix **X** contains more issue-commit instances encoding a true trace link. However, the generated feature matrices are still highly imbalanced (but less compared to 40% removal).

The augmentation classifier does benefit from increased number of true trace link instances. Comparing Figure 9.3 and Figure 9.2 the achieved measures are rather similar for all projects. In detail, the weighted precision slightly increases for most projects and reaches on average 99% for $\mathsf{RED}_{i,20\%}$ instead of on average 98% on RED_i. There is also a slight increase in averaged recall for all projects.

Lastly, the behavior of TLSA is evaluated using dataset $\mathsf{RED}_{100\%}$ where all existing issue-to-commit trace links have been removed. In this case only one dataset exists, because removing *all* issue-commit trace links does not involve randomness. The augmentation performance in terms of precision and recall[10] is shown in Figure 9.4. The classifier recommends a trace link for every issue-commit pair, resulting in perfect recall and worse precision for every project, of course. The exceptions are projects, which were highly linked in the first place, e.g. projects Hadoop, Hbase, or Hive (see Tab. 7.1). In this case, always recommending a link for a given issue-commit pair is

10. The classifier failed to distinguish the linked and nonlinked case, and therefore plotting weighted measures for pecision and recall is inappropriate.

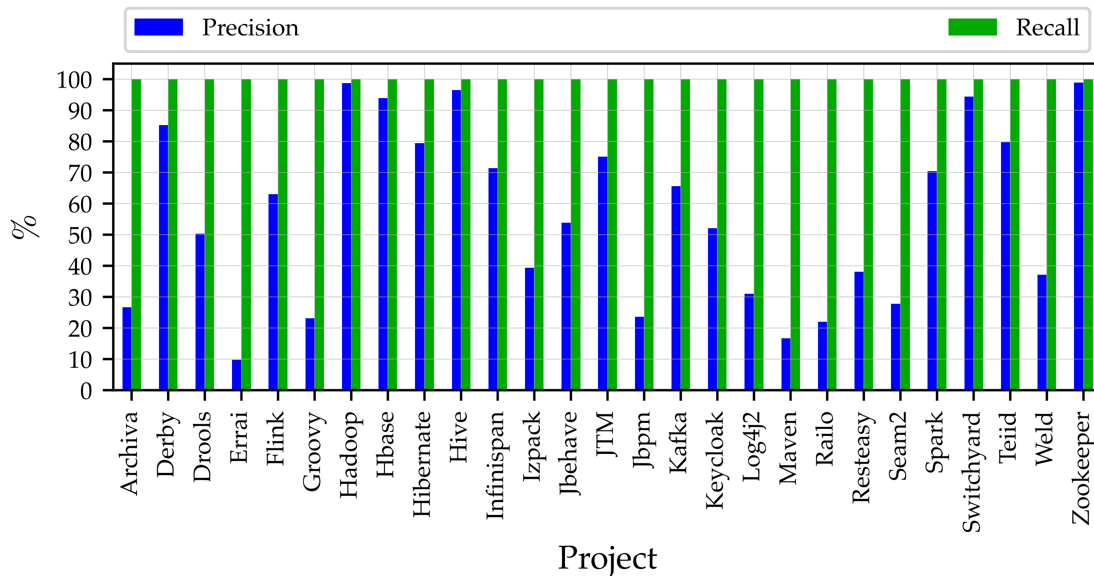


Figure 9.4.: Trace link set augmentation performance in terms of precision and recall (higher is better). Instead of removing 40% of issue-to-commit links, all issue-to-commit links are removed. The depicted measures are achieved on $\mathsf{RED}_{\text{red},100\%}$.

corrcect, of course. Indeed, precision nearly follows the linked issue-to-commit ratio of the projects. For example, project Seam2 has an unlinked issue-to-commit ratio of 72%, thus 28% are linked, which matches the achieved precision of 27.6% of the augmenter.

The reason for the poor augmentation results is the very limited feature data when creating the feature matrices for the projects in $\mathsf{RED}_{i,100\%}$. The features $\text{feat}_2 \ldots, \text{feat}_4$ of the process model describing relations to the closest linked commit are always 0.0, because of the absence of linked issue-commit pairs (see Sec. 7.4). The random forest classifier relies on these important features, and the stakeholder and similarity features alone do not permit good issue-to-commit trace link recovery.

Finding The Trace Link Set Augmentation classifier is able to recreate issue-to-commit trace links on projects with 40% nonlinked commits. The augmentation reliably reconstructs the correct links (barely false negatives), and only introduces few additional issue-to-commit trace links (false positives). The algorithm is project independent, and once trained can be used to recommend trace links on unknown projects. The classifier requires at least some existing issue-commit links, and cannot be applied to a project without any trace links.

9.5. RQ-5 - Effectiveness of IR-based Bug Localization Algorithms on Projects with Augmented Trace Link Sets

The bug localization algorithm ABLoTS outperforms AmaLgam on the augmented datasets AUG_i in terms of MAP (see Fig. 8.14) for all projects. On average MAP performance is 99% compared to reference dataset GS studied in experiment III. The largest difference is -0.03 MAP points for project Jbehave. Intrestingly, ABLoTS sometimes performs better on the augmented dataset, i.e. on project Archiva, or on project Hive, with about 2% better MAP. But, the Wilcoxon signed-rank test showes, that the MAP performance on GS is still significantly better ($p < 0.05$), but with negligible effect size (Cliffs $|\delta| = 0.018$). Nevertheless, the good performance on the datasets AUG_i shows, that bug localization using ABLoTS is still possible, even on projects containing incorrect issue-to-commit trace links (i.e. false positives introduced by the augmentation algorithm).

On average, the performance of AmaLgam in terms of MAP on AUG_i is identical to that achieved on gold standard GS. This is supported by the Wilcoxon signed-rank test, which cannot reject the null hypothesis, and thus there is no evidence, that the performance of AmaLgam on either dataset is different.

Changing the set of issue-commit links via removal or augmentation only affects the similar issue components, i.e. TraceScore in ABLoTS and SimiScore in AmaLgam, for the studied bug localization algorithms. The other components do not leverage this information. Thus any changes when comparing the achieved performance on GS and AUG_i stem from the performance of the similar issue component or the behavior of the composer component, which models to interactions between contained components (see Fig. 3.1). The later is difficult to investigate, especially for ABLoTS because its composer component is trained and no trivial formula is applied as for AmaLgam. The other possible reason can be investigated by comparing the performance of the similar issue components on gold standard GS and AUG_i as shown in Figure 9.5. It visualizes the achieved performance in terms of MAP macro averaged across the five datasets AUG_i labeled with “augmented”. For ease of comparision, the performance on reference data from experiment I is also added (see Fig. 8.3a).

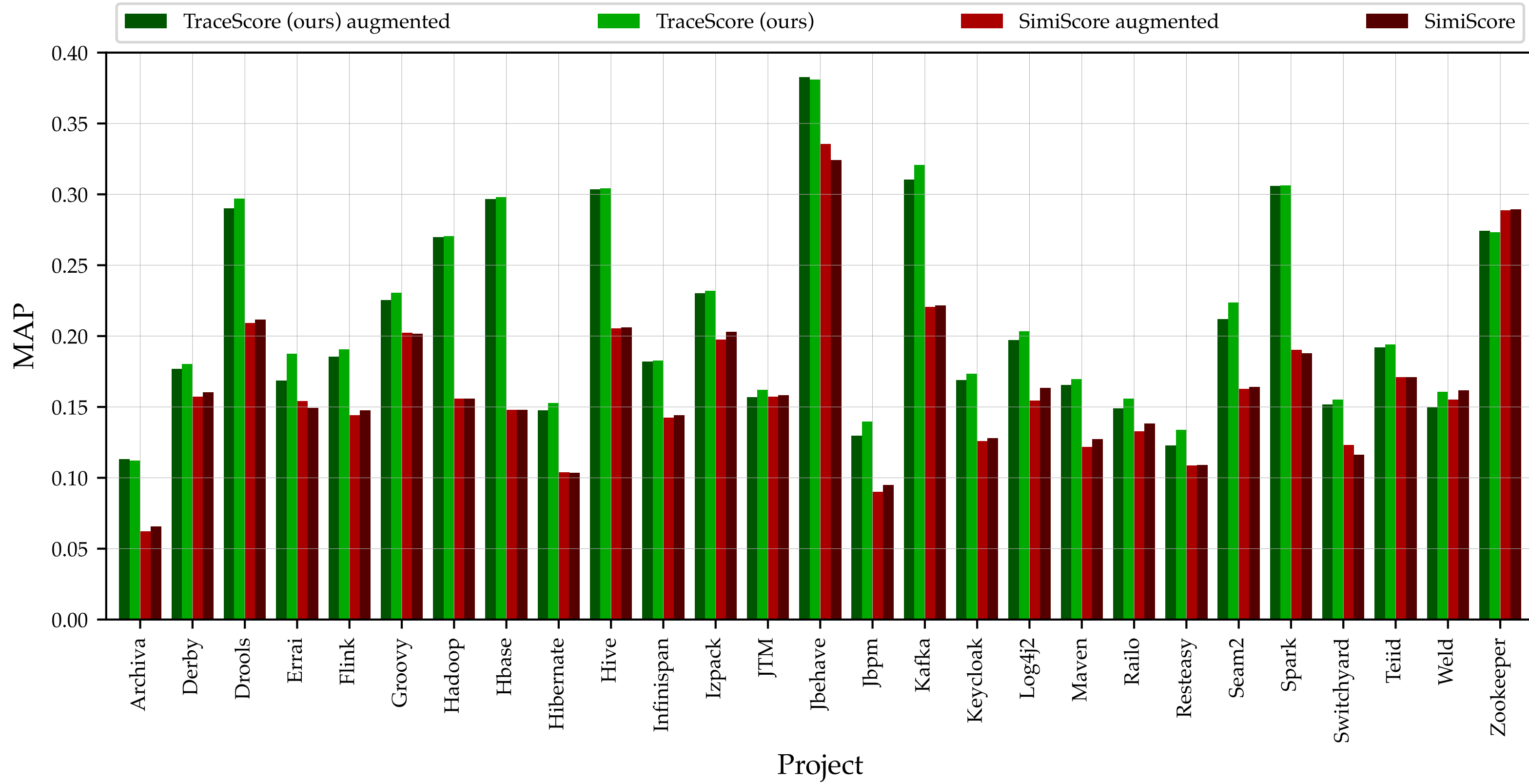


Figure 9.5.: Comparision of similar issue components in terms of MAP (higher is better). The bug localization is performed on AUG_i and averaged for TraceScore (ours) and SimiScore. For ease of comparision, the performance in terms of MAP from experiment I are depicted as well. The algorithms labeled ”augmented” were applied on issue-to-commit trace link augmented datasets, the others on reference GS.

The results of experiment IV showed varing precision values depending on the project, ranging from less than 10% to above 90% stemming from false positive issue-to-commit trace links. These should impact the similar issue component performance, because these links directly affect the calculation of SimiScore and TraceScore (see Eqs. 6.2, 6.6). Especially for TraceScore much larger traceability graphs are expected introducing many irrelevant source code files. However Figure 9.5 only shows minor changes in achieved MAP values, and TraceScore proofs robust to wrong issue-commit pairs. The applied traceability graph pruning controlled by parameters N_{bug} and N_{req} is responsible for this behaviour. It limits the inclusion of issues that modify too many source code files, resulting in similar traceability graphs created for GS and AUG_i. This situation is examplified in Figure 9.6.

Figure 9.6a shows a traceability graph created on reference GS for bug report b_{cur}. It contains two issues tracing to four source code files, i.e. the potential ones that need to be modified to resolve b_{cur}. The augmentation process introduces many false positive issue-to-commit trace links as experiment IV revealed. This is depicted in Figure 9.6b containing three augmented issue-to-commit trace links (b_0, c_2), (r_0, c_3), and (r_0, c_4), leading to 24 candidate source code files for b_{cur} (all highlighted in red). However, the TraceScore pruning process removes requirement r_0, because it traces to 21 source code files exceeding the baseline configuration limit of $N_{\text{req}} = 20$. The resulting traceability graph is shown in Figure 9.6c which finally is used in AUG_i, quite similar to the original one. Both traceability graphs share the candidate source code files f_0 and f_1. The files f_2, f_3 are only contained in the original graph, whereas the augmented and than pruned graph exclusively contains source code files f_4 and f_5. Thus, similar performance measures in terms of Top@k, AP, and RR for bug report b_{cur} in GS and AUG_i are expected.

The differences in number of source code files exclusively contained in reference traceability graph (green line) and the augmented and then pruned traceability graph (blue line) are visualized for projects Kafka and Spark in Figure 9.7. The x-axis represents the bug reports ordered by resolved date, i.e. 0 is the first (oldest) bug report, and that on far right the most recently resolved one. For example, the traceability graph created from the original data for the 100th bug report in project Spark contains 4 source code files, that are not present in the augmented and then pruned traceability graph. On the other hand, the augmented and then pruned traceability graph exclusively contains 13 source code files. Both graphs have 333 source code files in common (which is not depicted). For project Spark there are little differences between the two graphs, e.g. at max 10 files per bug report along the project lifetime (see Fig. 9.7a). Thus, the bug localization performance in terms of MAP on $\text{Proj}_{\text{Spark}}$ and $\text{Proj}_{\text{Spark,aug,i}}$ is expected to be similar, which is true (see Fig. 9.5). For project Kafka, the achieved MAP for TraceScore on $\text{Proj}_{\text{Kafka,aug},i}$ is 2 points worse than on gold standard $\text{Proj}_{\text{Kafka}}$. This stems from much higher differences of

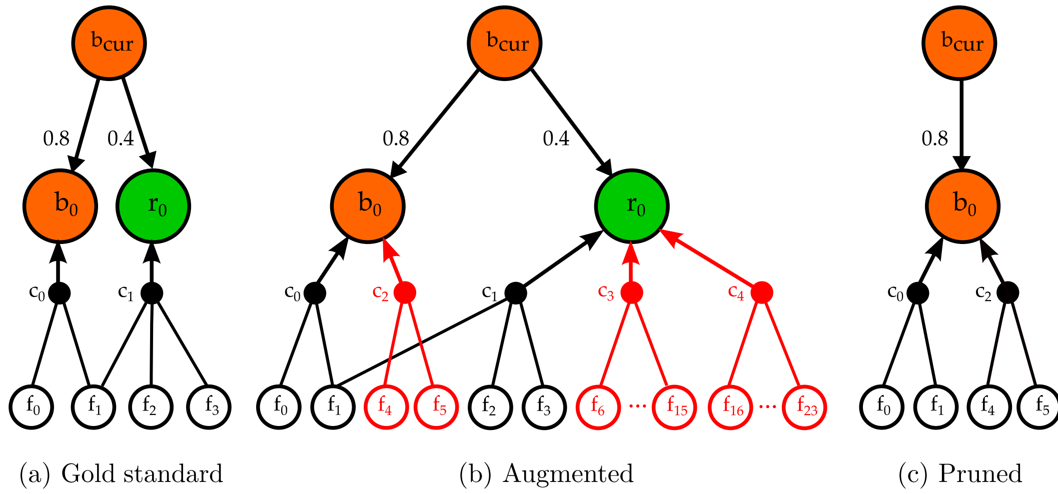
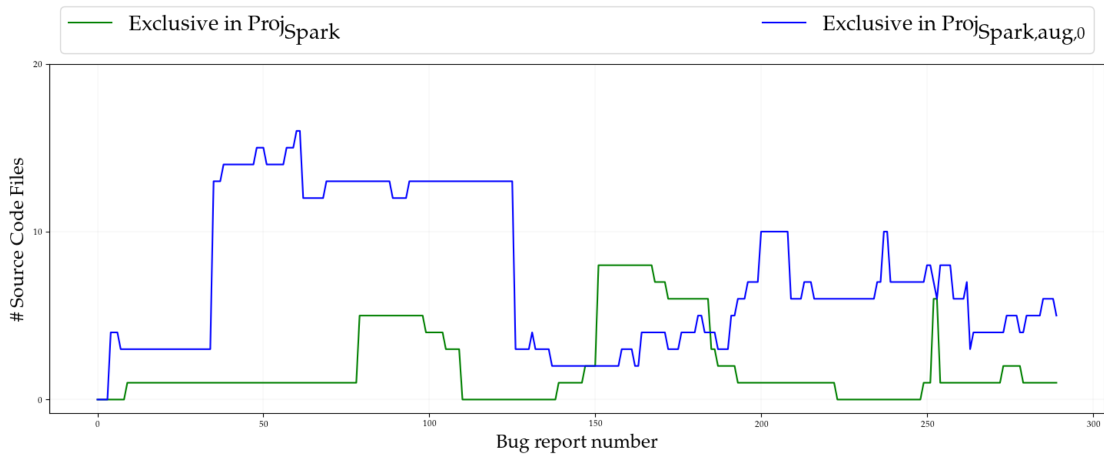
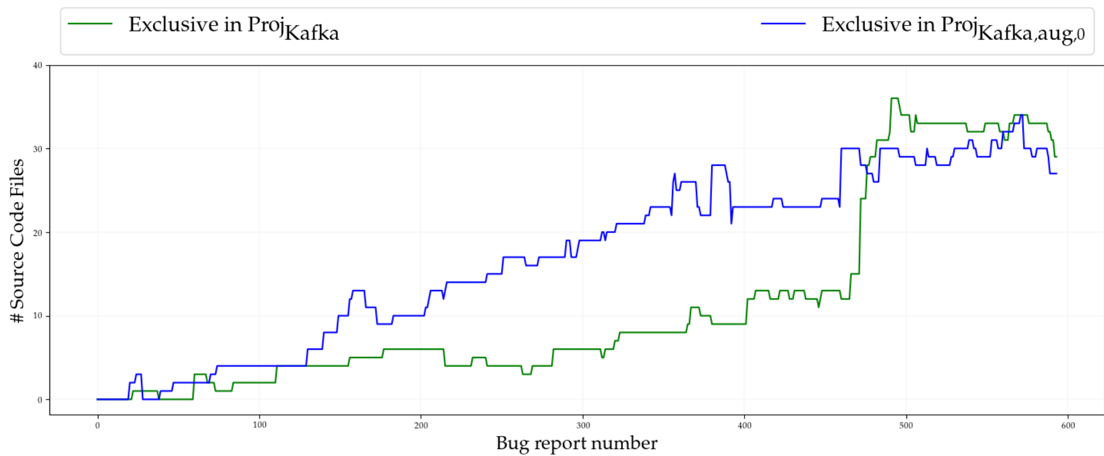


Figure 9.6.: Example for pruning of an augmented traceability graph. The left figure (a) shows the original traceablity graph for b_{cur} which traces to 4 source code files. The middle figure (b) shows the same graph, but with augmented issue-commit links (highlighted in red). Three additional (false positive) trace links (b_0, c_2), (r_0, c_3), and (r_0, c_4) have been added and b_{cur} traces to 24 source code files in total. The right figure (c) depicts the graph after pruning. Issue i_0 traces to 21 source code files, which exceeds $N_{\mathrm{req}} = 20$ of TraceScore baseline configuration. Thus r_0 is removed, and b_{cur} also traces to 4 source code files like the original. However the source code files differ. The figure uses the same legend as Figure 6.3.



(a) Project Spark



(b) Project Kafka

Figure 9.7.: Differences in number of source code files exclusively contained in the traceability graph in original project and those in trace link augmented project.

the constructed traceability graphs (see Fig. 9.7b). Following the projects lifetime, the exclusively contained source code files in either graph is continuously increasing, reaching about 30 source code files for the most recent bug report.

Finding The IR-based bug localization algorithm ABLoTS can be applied on projects with augmented issue-to-commit trace links with nearly no performance decrease. Its similar issue component TraceScore is robust against incorrectly added issue-to-commit trace links because of the applied traceability graph pruning process.

9.6. RQ-6 - Limitations of Studied Approaches

The last research question discusses the limitations of the developed approach, including the algorithms TraceScore, LuceneScore, ABLoTS, TLSA, and their interplay.

9.6.1. ABLoTS Requires a Project History

The novel IR-based bug localization algorithm ABLoTS is project specific. Further, its internal structure requires historic project data to train its composer component and thus being applicable. Therefore, ABLoTS cannot be used for new projects, i.e. it is technically not possible to locate defective source code files for the first bug reports. Further, it is currently unknown how much historical data is required for training ABLoTS to provide acceptable results. The chosen time based $80\% - 20\%$ split of bug reports in experiment III (see Sec. 8.6) results in training history lengths, i.e. the time between the first train bug report was created and the last training bug report was resolved, ranging from 1,024 days in project Flink, up to 3,736 days in project Maven ($\approx$ 2,400 days on average).

9.6.2. Augmenting Issue-to-Commit Trace Link Set in Large Projects

There is a limitation concerning the augmentation of trace links sets for certain projects. For example, the projects Cassandra, Lucene, and Wildfly had to be excluded during the construction of GS out of SEOSS. The feature creation routine of TLSA algorithm (see Sec. 7.4.3) could not handle these projects. This routine of creating issue-commit pairs depends on the amount of respective artifacts present in a project. However, comparing the number of issues and commits in project Cassandra and Hadoop (see Tab. 5.2), project Cassandra contains fewer numbers in both figures.

Nevertheless, project Cassandra is more difficult and currently infeasible to process. The issue-commit pair creation also depends on temporal relationships. In case a commit is filed when lots of unresolved issues exists, this commit is subject to be traced to all of these issues. For example, in Figure 7.2 the commit c_7 is a candidate to be traced to requirements r_2, r_3 and bug report b_0, and respective feature vectors are calculated. The more issues are in active development, i.e. not in status CLOSED or RESOLVED, when a commit occurs, the more issue-commit pairs are possible. These temporal relations exaggerate the amount for reasonable issue-commit pairs for projects Cassandra, Lucene, and Wildfly, although the amount of issues and commits in these projects is less than as that of project Hadoop, which is processed unproblematically. Thus, the current *valid* constraints based on casuality are not sufficient to handle these edge cases. Further investigations are required.

9.6.3. Issue-to-Commit Trace Link Set Augmentation Requires Existing Links

The TLSA algorithm is project independent. However, it can not be applied to a project containing no issue-to-commit trace links at all, as a dedicated study in experiment IV revealed (see Sec. 9.4). In case no issue-to-commit links exist, the crucial process related features $\text{feat}_2, \ldots, \text{feat}_4$ cannot be calculated and thus are all set to 0.0 (see Sec. 7.4). These features are designed to detect bursts of consequitive commits, and only one (or few) are traced to an issue. However, these commits still share similarities with respect to temporal closeness and resource overlap, even in case none of them is explicitly linked to an issue. Eleborate strategies are required to handle these cases and thus assign purposeful values for features $\text{feat}_2, \ldots, \text{feat}_4$.

9.7. Threats to Validity

There are several potential threats to the validity of applied methods and conducted experiments in this thesis. The discussion includes four common categories (Runeson and Höst 2009; Yin 2009). *Construct validity* reflects to which extent the operational measures that are studied represent what is investigated according to research questions. *Internal validity* is of concern when investigating casual relations, i.e. to what extent a piece of evidence supports a claim about cause and effect. *External validity* deals with the possibility to generalize findings outside of the context of a study. Last, *reliability* is concerned with to what extent the analysis and data are dependent on researchers that conducted a study.

The discussion is separated in several sections each concerned with a specific topic of the thesis. For each topic, the potential threats to validity and how they were mitigated are reviewed.

9.7.1. Project Selection and Dataset Creation

Construct Validity The analyzed trace links were created manually by project members in all analyzed projects. This implies the risk that semantically incorrect trace links were created or trace links were forgotten by mistake. Many projects outside of safety critical domains do not have reliable traceability information (Cleland-Huang et al. 2014). The following three aspects indicate a very high trace link quality in all projects. First, all projects' quality assurance process is based on the created trace links. The projects established a manual process where changes are reviewed and tested by humans. All 33 projects have in common that the quality of the established trace links is implicitly verified through this process. Second, the explicit change approval process in all 33 projects ensures that the four-eyes-principle is applied for each manually created trace link. Third, the openness of all 33 projects (all development artifacts are publicly available) enables anyone to participate in the project and review the created trace links. Due to these facts, the author considers the risk of incorrect or forgotten trace links is sufficiently mitigated.

Internal Validity The issue artifacts were mined from projects' ITS. The used issue type, bug report or requirement, is subject to misclassification (Herzig, Just, and Zeller 2013), i.e. a bug report is actually a feature or vice versa. Indeed, misclassification affects bug localization, but the effect size is negligible (Kochhar, Tian, and Lo 2014). Table 8.1 shows that the median values of modified source code files for bug reports are lower than that for requirements. This indicates, that the bug reports in the golden set GS are not inflated.

The construction of golden set GS explicitly only considers specific issue artifacts, i.e. those having a status RESOLVED or CLOSED. The assumption is, that all necessary source code modifications had taken place. To mitigate this threat, the respective issue resolution was also taken into account and needs to be set to FIXED or DONE.

A guided, random issue-to-commit trace link removal process was performed to derive reduced datasets RED_i from GS. To mitigate the random effects, this process was repeated five times with different seed values for the random number generator. All following experiments were separately performed on each dataset RED_i and results were averaged. However, choosing a repetition rate other than five might result in different averaged metrics.

External Validity All selected projects are open-source software, since those were the only available projects that provided all the necessary information to conduct the studies. Generalizing the findings to a wider population including commercial projects poses a potential threat to external validity. However, the applied software lifecycle management tools Atlassian Jira and Git are also a popular combination of tools within closed-source projects. Actually, Atlassian provides and sells *Bitbucket*, a Git-based source code repository hosting service similar to GitHub. Jira and Bitbucket have a tight integration[11]. In open-source and closed-source project development agile methodologies are widely adopted. Due to the similarities, there is evidence to generalize the findings to a large population of closed-source projects. However, replications of the conducted study with commercial projects are required to justify this assumption.

The project selection only includes those using Jira as ITS and Git as VSC, which raises another threat to external validity. This specific tool configuration was chosen based on popularity polls, which also showed that other tool combinations are available and actively used. These tools and platforms might provide or encourage a different trace link creation behavior.

Reliability The 33 projects mined for the SEOSS dataset were selected by the author. This selection might be biased due to certain experiences or preferences. To mitigate this threat, a set of selection criteria were defined upfront. Further, the applied maximum variation case strategy ensures to draw representative samples based on project's characteristics. However, other researchers could have selected other projects.

Another potential threat exists in the collection and preparation of the project data. To avoid especially manual bias during project data preparation and to ensure reproducible results, a fully automated data mining process was used. Due to the public availability of the project artifacts and the fully automated collection and analysis process, it can be replicated and additional projects could be included to further broaden the data corpus. The used tooling to automate this process was carefully verified. Therefore, intermediate results were manually validated and continuously cross-checked for inconsistencies and contradictions.

9.7.2. Experiments I-V

External Validity All reported results are only valid for the 27 example projects. Conducting the experiments on another set of projects may provide different results. The initial project selection tries to mitigate this threat, by selecting a wide variety of projects with different characteristics also including edge cases.

11. https://bitbucket.org/product/integrations

9.7.3. Specifically for Experiments I and II

Internal Validity The TraceScore algorithm has four parameters to filter artifacts based on amount of modified source code files and time. A baseline set of values for the parameters were given, claiming to provide good project independent performance. To mitigate this threat, additional configurations have been evaluated, but not exhaustively and better configurations might exist.

9.7.4. Specifically for Experiment III

Internal Validity The composer component of ABLoTS is a binary classifier. This requires to be trained and therefore the available data for each project was split into $80\% - 20\%$ of the bug reports retaining the temporal ordering. Choosing another split point may produce different evaluation results.

The training data for ABLoTS is severely imbalanced. Thus it was randomly subsampled to create balanced data. To mitigate the random effects, training and testing was repeated five times and achieved performance results are averages of the individual runs.

9.7.5. Specifically for Experiment IV

Internal Validity The TLSA algorithm also uses a binary classifier, like the ABLoTS composer component. Again, the training data is highly imbalanced, and was randomly subsampled to create balanced data. To mitigate the random effects, training and testing was repeated five times. The achieved results were derived based on a majority vote, i.e. at least three out of the five results need to agree.

10. Conclusion and Future Work

This section summarizes the conclusions that can be drawn from this thesis. Research possibly never ends and there is always potential to proceed and improve. Thus further research directions are also outlined.

10.1. Summary

This thesis contributes to the body of knowledge in bug localization, automatic trace link creation and software maintenance. The hypothesis of this dissertation is that bug localization performance benefits by integrating additional traced artifacts, i.e. a project's requirements and existing trace links, especially those from bug reports and requirements to source code files. Therefore a holistic approach was proposed to achieve this goal.

Section 4 outlined the approach and introduced an artifact model and its mathematical foundation used throughout this thesis.

Reviewing the state of the art revealed, that no suitable dataset containing the required artifacts and trace links to explore novel algorithms exists. Further, existing datasets used to evaluate state-of-the-art bug localization algorithms are rather small. Therefore, Section 5 introduced a mining process to create the SEOSS dataset consisting of 33 open-source projects. It contains 600,000 artifacts and 300,000 trace links collected by analyzing projects' issue tracking systems and version control systems. All gathered data is stored in relational databases, that allow a convenient access to query relevant information. The complete projects' lifecycle is captured, enables empirical studies, and is not limited to the field of bug localization.

Section 6 gave an in-depth discussion of IR-based bug localization algorithms. First, the internal architecture of two algorithms were studied to exploit requirement and trace link information. The similar issue component, a component found in many IR-based bug localization algorithms, was identified as candidate to leverage traceability information. This resulted in the design of the similar issue component *TraceScore*. It utilizes information of previously implemented requirements, resolved bug reports, and trace links among them to suggest candidate source code files that are relevant to fix a current bug report at hand. TraceScore is one of many components including

the source code component and the history component of a modern IR-based bug localization algorithm. Therefore a novel algorithm *ABLoTS* was created utilizing TraceScore. ABLoTS also incorporates a novel component, *LuceneScore*, to analyze the structure of a project's source code. LuceneScore is an improved version of a source code structure component studied at the beginning of Section 6.

The SEOSS dataset contains hundreds of thousands trace links from requirement to source code artifacts. However, a study conducted in Section 7 revealed, that on average 40% of a project's commit artifacts are not traced to any requirement or bug report. Reducing this amount by establishing new trace links has the potential to improve ABLoTS' bug localization performance, because its contained TraceScore component is able to utilize trace link information. Therefore the typical development workflow of an agile software project was studied. Three models were derived capturing essentials aspects of the workflow. Leveraging these, eleven features were identified that describe a trace link between issue artifacts and commit artifacts. Last, the novel *TLSA* algorithm based on machine learning techniques was designed to automatically augment the project's existing set of issue-to-commit trace links.

All developed algorithms were exhaustively evaluated in Section 8 and the achieved results were discussed in Section 9. The evaluation was performed on a curated subset of the previously created SEOSS dataset. Five experiments were conducted each answering a specific research question. The first experiment questioned the effectiveness of TraceScore compared to two other similar issue components. TraceScore achieves on average a 53% higher Top-1, and 33% better MAP as its closest competitor. The second experiment studied different parameter configurations of TraceScore. The result was a baseline configuration suitable for all evaluated projects, providing a starting point when used for new projects. Next, an experiment was presented comparing the bug localization performance of ABLoTS compared to three other algorithms. ABLoTS outperformed the closest competitors by 100% in terms of Top@k, MAP, and MRR on average. Therefore, leveraging requirement artifacts and trace link information proofs its effectiveness. The fourth experiment studied the automatic issue-to-commit trace link augmentation using the novel TLSA algorithm. It showed, that it is possible to correctly augment the majority of missing trace links. Further, the created TLSA algorithm is *project independent* and thus can be applied to a project out of the box. The last experiment attempted to answer, whether IR-based bug localization benefits from augmented issue-to-commit trace links sets. The achieved performance for ABLoTS algorithm is comparable to those when all issue-to-commit trace links were manually created and maintained.

10.2. Future Work

The developed IR-based bug localization algorithm ABLoTS takes a bug report and queries the project's code base to identify a list of candidate source code files that need to be modified in order to fix the bug. In this IR-based approach, a query is formulated using the bug report's textual elements and applying a common text pre-processing pipeline. This pipeline is applied to the bug report's texts' regardless of the content (assuming it is natural language). No differences of the actual content are made, and the evaluation in this thesis showed the effectiveness of this approach. However, Wang et al. reported the performance of IR-based algorithms heavily relies on the usefulness of the pre-processed queries (Wang, Parnin, and Orso 2015). Previous studies showed that bug report descriptions may contain additional *rich* information such as stack traces and source code snippets next to natural language (Bettenburg et al. 2008; Moreno et al. 2014). A study on seven open-source projects provides evidence, that the presence of this additional information affects the performance of IR-based algorithms, but the effect size is small (Rath and Mäder 2019a). The authors analyzed the structure of bug reports' descriptions and applied different query formulation strategies based on found information. It would be interesting to also integrate this regime to ABLoTS and study its impact on bug localization performance.

The performed experiments are analytical, which is common in many studies concerning bug localization. The localization performance is measured by evaluating whether defective source code files appear on top of the generated ranked source code file lists. However, researchers questioned the usefulness of these lists to aid the developers in locating the bug (Wang, Parnin, and Orso 2015). Further, the authors argue that the source code file level is to coarse-grained and still leaves the developers with a large amount of code (in terms of lines) to examine. Wang et al. suggest to conduct user studies to investigate the effectiveness of ranked source code file lists. Thus, conducting a user study involving project developers to evaluate the approaches of this thesis is a potential future work. One step in this direction was already performed by Rath et al. (Rath and Mäder 2019a). In this work, a user study was conducted and the participants should identify the defective source code file based on a given ranked list. The authors also proposed a visualization technique to assist the participants during this task. Thus, highlighted source code snippets of the files were provided next to the ranked list of source code file names. The applied semantic highlighting is based on the bug report's text and matching tokens found in the source code and thus provides more details and finer grained bug locations. The evaluation showed that the combination of ranked lists and the visualization technique is beneficial to distinguish false positive source code file recommendations from true ones. A different approach to address the coarse granularity of files was proposed by Wen et al. (Wen, Wu, and Cheung 2016). They argue, that the software

change level, i.e. single commits, is better suited than the source code file level. Extending ABLoTS with this functionality provides an interesting challenge. Instead of generating a list of source code files, a list of defective commits could be presented to the responsible developer.

Another avenue for future work is to extend the evaluation to a large scale of projects including commercial ones. From a tooling perspective, the approaches developed in this thesis should be applicable, because the combination of ITS Jira and VCS Git are used in commercially developed software as well. To simplify the deployment process, the developed approaches ABLoTS and TLSA should be combined to a prototype. A first step in this direction has already been made. The recommendation system `spojitR` (Rath, Tomova, and Mäder 2020) partially implements TLSA and a prototype is available[1]. It tightly integrates into the commit process, and reminds the developer to tag each commit message. Therefore it proposes the three most suitable issues from the ITS. This enables the developer to easily choose the correct one and the commit is automatically tagged. The developed tool could also be extended to retrospectively augment missing issue-to-commit trace links.

1. https://github.com/SECSY-Group/spojitr

A. Appendix

A.1. Evaluating Different Temporal Settings for TraceScore

The Figures A.1, A.2 show the performance of TraceScore for different settings for the history length parameters D_{bug} and D_{req} in terms of MAP and MRR as outlined in Section 8.4. The x-axis shows the project, and the y-axis the respective measure: MAP or MRR. The configuration is color coded.

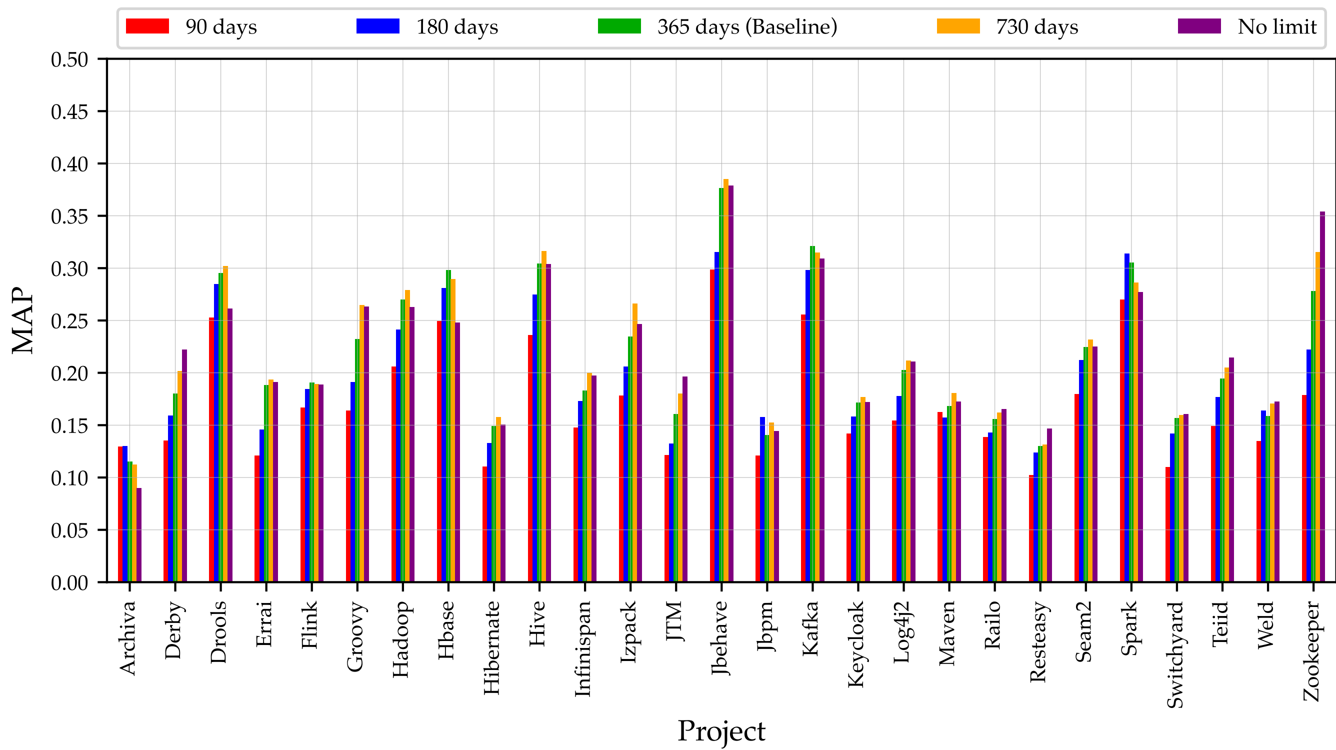


Figure A.1.: Comparision of different history lengths ($D_{\text{bug}} = D_{\text{req}}$) for TraceScore in terms of MAP (higher is better).

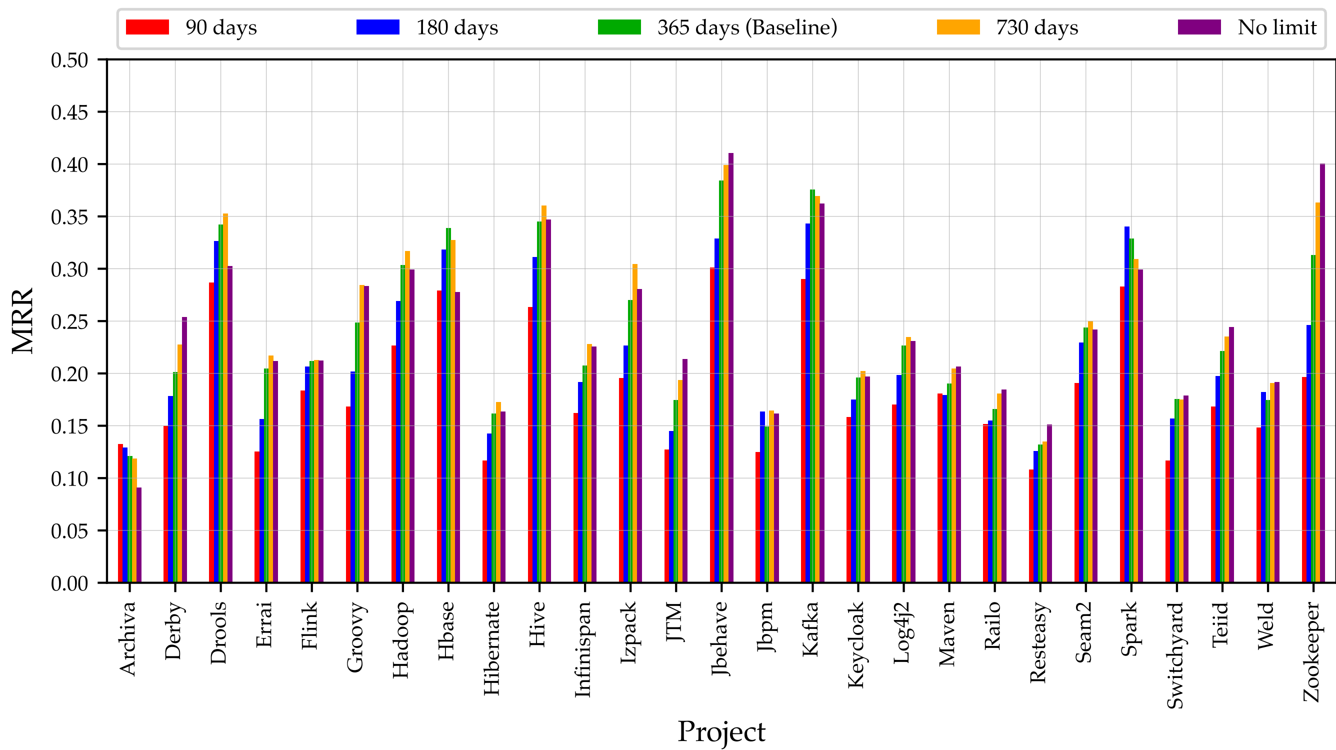


Figure A.2.: Comparision of different history lengths ($D_{\text{bug}} = D_{\text{req}}$) for TraceScore in terms of MRR (higher is better).

A.2. Evaluation Details for Similar Issue Components

Table A.1 depicts details of the performance of different similar issue components in terms of Top@k, MAP, and MRR as described in Section 8.4. Each row represent a projects, sub divided by evaluated similar issue component. The columns are the different achieved performance metrics. The top performance measure (higher is better) per project is highlighted using bold font.

Table A.1.: Comparison of different similar issue components in terms of Top@1, Top@5, Top@10, MAP, and MRR (higher values are better, highest value is bold).

Project	Algorithm	Top@1	Top@5	Top@10	MAP	MRR
Archiva	CollabScore	0.071	0.107	0.107	0.068	0.081
	SimiScore	0.024	0.107	0.131	0.066	0.066
	TraceScore (ours)	**0.085**	**0.110**	**0.256**	**0.112**	**0.121**
Derby	CollabScore	0.052	0.112	0.147	0.079	0.082
	SimiScore	0.109	0.259	0.345	0.160	0.181
	TraceScore (ours)	**0.128**	**0.281**	**0.368**	**0.180**	**0.202**
Drools	CollabScore	0.059	0.139	0.186	0.092	0.100
	SimiScore	0.136	0.371	0.477	0.212	0.248
	TraceScore (ours)	**0.239**	**0.478**	**0.572**	**0.297**	**0.345**
Errai	CollabScore	0.034	0.086	0.120	0.059	0.061
	SimiScore	0.094	0.240	0.352	0.149	0.165
	TraceScore (ours)	**0.139**	**0.274**	**0.361**	**0.187**	**0.202**
Flink	CollabScore	0.059	0.132	0.153	0.085	0.089
	SimiScore	0.104	0.245	0.303	0.147	0.168
	TraceScore (ours)	**0.134**	**0.298**	**0.375**	**0.191**	**0.212**
Groovy	CollabScore	0.054	0.111	0.123	0.075	0.079
	SimiScore	0.118	0.323	0.422	0.201	0.218
	TraceScore (ours)	**0.159**	**0.349**	**0.428**	**0.231**	**0.247**
Hadoop	CollabScore	0.073	0.139	0.166	0.098	0.103
	SimiScore	0.087	0.273	0.353	0.156	0.175
	TraceScore (ours)	**0.210**	**0.419**	**0.503**	**0.270**	**0.304**
Hbase	CollabScore	0.051	0.113	0.147	0.078	0.082
	SimiScore	0.089	0.241	0.338	0.148	0.165
	TraceScore (ours)	**0.233**	**0.472**	**0.563**	**0.298**	**0.339**
Hibernate	CollabScore	0.032	0.059	0.074	0.043	0.044
	SimiScore	0.065	0.160	0.210	0.104	0.112
	TraceScore (ours)	**0.110**	**0.218**	**0.292**	**0.153**	**0.165**
Hive	CollabScore	0.073	0.177	0.236	0.112	0.123
	SimiScore	0.124	0.364	0.464	0.206	0.237

	TraceScore (ours)	**0.238**	**0.475**	**0.570**	**0.304**	**0.345**
Infinispan	CollabScore	0.053	0.113	0.142	0.078	0.082
	SimiScore	0.098	0.239	0.314	0.144	0.166
	TraceScore (ours)	**0.129**	**0.295**	**0.383**	**0.183**	**0.207**
Izpack	CollabScore	0.077	0.140	0.174	0.096	0.107
	SimiScore	0.165	0.322	0.397	0.203	0.238
	TraceScore (ours)	**0.187**	**0.358**	**0.457**	**0.232**	**0.268**
JTM	CollabScore	0.042	0.114	0.133	0.070	0.073
	SimiScore	0.117	0.211	**0.292**	0.158	0.167
	TraceScore (ours)	**0.133**	**0.231**	0.286	**0.162**	**0.177**
Jbehave	CollabScore	0.080	0.100	0.100	0.088	0.088
	SimiScore	0.260	0.440	0.520	0.324	0.334
	TraceScore (ours)	**0.300**	**0.540**	**0.580**	**0.381**	**0.388**
Jbpm	CollabScore	0.043	0.082	0.091	0.057	0.062
	SimiScore	0.058	0.165	0.216	0.095	0.106
	TraceScore (ours)	**0.074**	**0.225**	**0.320**	**0.140**	**0.148**
Kafka	CollabScore	0.082	0.184	0.261	0.120	0.135
	SimiScore	0.157	0.399	0.481	0.221	0.265
	TraceScore (ours)	**0.254**	**0.524**	**0.611**	**0.321**	**0.376**
Keycloak	CollabScore	0.031	0.077	0.102	0.051	0.053
	SimiScore	0.084	0.207	0.294	0.128	0.144
	TraceScore (ours)	**0.123**	**0.276**	**0.353**	**0.173**	**0.199**
Log4j2	CollabScore	0.051	0.086	0.107	0.066	0.069
	SimiScore	0.113	0.248	0.334	0.163	0.180
	TraceScore (ours)	**0.154**	**0.314**	**0.388**	**0.203**	**0.227**
Maven	CollabScore	0.040	0.106	0.139	0.058	0.070
	SimiScore	0.093	0.219	0.278	0.127	0.149
	TraceScore (ours)	**0.126**	**0.259**	**0.364**	**0.169**	**0.193**
Railo	CollabScore	0.023	0.093	0.133	0.054	0.061
	SimiScore	**0.100**	0.213	0.300	0.138	0.157
	TraceScore (ours)	0.094	**0.246**	**0.323**	**0.156**	**0.166**
Resteasy	CollabScore	0.040	0.084	0.096	0.056	0.058
	SimiScore	0.056	0.183	**0.242**	0.109	0.111
	TraceScore (ours)	**0.088**	**0.200**	0.241	**0.134**	**0.137**
Seam2	CollabScore	0.039	0.107	0.135	0.069	0.073
	SimiScore	0.115	0.244	0.313	0.164	0.174
	TraceScore (ours)	**0.163**	**0.342**	**0.414**	**0.223**	**0.243**
Spark	CollabScore	0.038	0.121	0.155	0.070	0.076
	SimiScore	0.110	0.310	0.376	0.188	0.199
	TraceScore (ours)	**0.221**	**0.464**	**0.567**	**0.306**	**0.330**
Switchyard	CollabScore	0.041	0.065	0.087	0.051	0.055
	SimiScore	0.084	0.183	0.239	0.116	0.131

	TraceScore (ours)	**0.113**	**0.224**	**0.311**	**0.155**	**0.172**
Teiid	CollabScore	0.040	0.079	0.100	0.057	0.059
	SimiScore	0.115	0.281	0.377	0.171	0.194
	TraceScore (ours)	**0.135**	**0.317**	**0.404**	**0.194**	**0.222**
Weld	CollabScore	0.018	0.063	0.075	0.035	0.036
	SimiScore	**0.124**	0.231	0.315	**0.162**	**0.186**
	TraceScore (ours)	0.105	**0.251**	**0.335**	0.161	0.177
Zookeeper	CollabScore	0.103	0.215	0.276	0.143	0.162
	SimiScore	**0.221**	**0.491**	**0.605**	**0.289**	**0.340**
	TraceScore (ours)	0.199	0.438	0.524	0.273	0.309

A.3. Evaluation Details for Bug Localization Algorithms

Table A.2 shows details of the performance of different bug localization algorithms terms of Top@k, MAP, and MRR, as described in Section 8.6. Each row represent a projects, sub divided by bug localization algorithm. The columns are the different achieved performance metrics. The top performance measure (higher is better) per project is highlighted using bold font.

Table A.2.: Comparison of different bug localization algorithms in terms of Top@1, Top@5, Top@10, MAP, and MRR on testing dataset (higher values are better, highest value is bold).

Project	Algorithm	Top@1	Top@5	Top@10	MAP	MRR
Archiva	ABLoTS (ours)	**0.200**	**0.388**	**0.506**	**0.232**	**0.299**
	AmaLgam	0.059	0.235	0.353	0.142	0.160
	BLUiR	0.059	0.235	0.353	0.139	0.157
	LuceneScore (ours)	0.176	0.353	0.412	0.207	0.260
Derby	ABLoTS (ours)	**0.418**	**0.635**	**0.706**	**0.463**	**0.515**
	AmaLgam	0.263	0.445	0.552	0.309	0.350
	BLUiR	0.232	0.403	0.538	0.284	0.322
	LuceneScore (ours)	0.398	0.613	0.692	0.436	0.494
Drools	ABLoTS (ours)	**0.270**	**0.484**	**0.570**	**0.313**	**0.368**
	AmaLgam	0.165	0.299	0.396	0.192	0.230
	BLUiR	0.144	0.277	0.367	0.174	0.210
	LuceneScore (ours)	0.180	0.385	0.475	0.234	0.276
Errai	ABLoTS (ours)	**0.306**	**0.549**	**0.647**	**0.387**	**0.419**
	AmaLgam	0.191	0.426	0.511	0.261	0.294
	BLUiR	0.170	0.426	0.511	0.258	0.282

	LuceneScore (ours)	0.277	0.511	0.617	0.361	0.393
Flink	ABLoTS (ours)	**0.512**	**0.728**	**0.772**	**0.532**	**0.602**
	AmaLgam	0.298	0.508	0.562	0.345	0.390
	BLUiR	0.295	0.488	0.558	0.340	0.384
	LuceneScore (ours)	0.500	0.705	0.764	0.517	0.586
Groovy	ABLoTS (ours)	**0.381**	**0.673**	**0.732**	**0.466**	**0.505**
	AmaLgam	0.247	0.353	0.459	0.271	0.296
	BLUiR	0.235	0.353	0.435	0.267	0.288
	LuceneScore (ours)	0.329	0.612	0.718	0.411	0.459
Hadoop	ABLoTS (ours)	**0.536**	**0.741**	**0.795**	**0.566**	**0.626**
	AmaLgam	0.188	0.342	0.412	0.237	0.259
	BLUiR	0.158	0.319	0.395	0.213	0.234
	LuceneScore (ours)	0.473	0.684	0.746	0.513	0.568
Hbase	ABLoTS (ours)	**0.525**	**0.758**	**0.818**	**0.560**	**0.625**
	AmaLgam	0.204	0.395	0.489	0.264	0.293
	BLUiR	0.199	0.370	0.442	0.247	0.274
	LuceneScore (ours)	0.455	0.696	0.777	0.504	0.565
Hibernate	ABLoTS (ours)	**0.300**	**0.514**	**0.571**	**0.362**	**0.393**
	AmaLgam	0.044	0.100	0.128	0.065	0.069
	BLUiR	0.037	0.078	0.106	0.056	0.060
	LuceneScore (ours)	0.221	0.393	0.492	0.283	0.308
Hive	ABLoTS (ours)	**0.435**	**0.689**	**0.747**	**0.486**	**0.546**
	AmaLgam	0.257	0.495	0.579	0.320	0.363
	BLUiR	0.219	0.450	0.520	0.286	0.324
	LuceneScore (ours)	0.351	0.601	0.686	0.420	0.465
Infinispan	ABLoTS (ours)	**0.327**	**0.499**	**0.573**	**0.364**	**0.407**
	AmaLgam	0.145	0.303	0.393	0.191	0.219
	BLUiR	0.141	0.288	0.367	0.185	0.213
	LuceneScore (ours)	0.316	0.444	0.519	0.337	0.383
Izpack	ABLoTS (ours)	**0.397**	**0.696**	**0.748**	**0.464**	**0.527**
	AmaLgam	0.164	0.247	0.260	0.191	0.203
	BLUiR	0.137	0.219	0.247	0.155	0.168
	LuceneScore (ours)	0.342	0.644	0.712	0.433	0.476
JTM	ABLoTS (ours)	**0.297**	**0.574**	**0.684**	**0.397**	**0.422**
	AmaLgam	0.226	0.500	0.581	0.326	0.342
	BLUiR	0.226	0.500	0.581	0.313	0.338
	LuceneScore (ours)	0.258	0.548	0.661	0.371	0.396
Jbehave	ABLoTS (ours)	**0.580**	0.780	0.820	**0.627**	**0.680**
	AmaLgam	0.300	0.600	0.800	0.409	0.456
	BLUiR	0.300	0.700	0.800	0.390	0.439
	LuceneScore (ours)	0.500	**0.800**	**0.900**	0.586	0.637
Jbpm	ABLoTS (ours)	**0.148**	**0.309**	0.345	**0.184**	**0.222**
	AmaLgam	0.106	0.273	**0.394**	0.150	0.185
	BLUiR	0.106	0.273	0.379	0.150	0.186

	LuceneScore (ours)	0.121	0.288	0.333	0.151	0.195
Kafka	ABLoTS (ours)	**0.514**	**0.751**	**0.808**	**0.537**	**0.619**
	AmaLgam	0.235	0.504	0.588	0.303	0.351
	BLUiR	0.218	0.437	0.563	0.296	0.334
	LuceneScore (ours)	0.445	0.681	0.765	0.461	0.554
Keycloak	ABLoTS (ours)	**0.285**	**0.592**	**0.676**	**0.366**	**0.415**
	AmaLgam	0.143	0.317	0.402	0.190	0.226
	BLUiR	0.138	0.317	0.397	0.189	0.223
	LuceneScore (ours)	0.206	0.460	0.550	0.287	0.330
Log4j2	ABLoTS (ours)	**0.504**	**0.731**	0.792	**0.537**	**0.599**
	AmaLgam	0.255	0.439	0.480	0.310	0.333
	BLUiR	0.245	0.429	0.480	0.296	0.320
	LuceneScore (ours)	0.490	0.704	**0.796**	0.521	0.596
Maven	ABLoTS (ours)	**0.232**	**0.355**	**0.471**	**0.279**	**0.303**
	AmaLgam	0.161	0.258	0.387	0.190	0.228
	BLUiR	0.161	0.258	0.387	0.190	0.227
	LuceneScore (ours)	0.226	0.355	0.452	0.266	0.294
Railo	ABLoTS (ours)	**0.250**	**0.533**	**0.580**	**0.332**	**0.367**
	AmaLgam	0.200	0.483	0.517	0.279	0.323
	BLUiR	0.183	0.483	0.517	0.272	0.310
	LuceneScore (ours)	0.233	0.450	0.517	0.315	0.337
Resteasy	ABLoTS (ours)	**0.314**	**0.526**	**0.588**	**0.363**	**0.403**
	AmaLgam	0.031	0.092	0.123	0.057	0.061
	BLUiR	0.031	0.108	0.123	0.060	0.063
	LuceneScore (ours)	0.292	0.523	0.585	0.338	0.381
Seam2	ABLoTS (ours)	**0.312**	**0.479**	**0.544**	**0.372**	**0.389**
	AmaLgam	0.083	0.160	0.192	0.110	0.117
	BLUiR	0.077	0.155	0.187	0.105	0.110
	LuceneScore (ours)	0.282	0.442	0.500	0.343	0.355
Spark	ABLoTS (ours)	0.438	**0.662**	0.700	**0.493**	0.537
	AmaLgam	0.293	0.621	**0.724**	0.408	0.448
	BLUiR	0.293	0.586	0.724	0.401	0.441
	LuceneScore (ours)	**0.483**	0.638	0.707	0.481	**0.556**
Switchyard	ABLoTS (ours)	**0.096**	**0.227**	**0.308**	**0.147**	**0.158**
	AmaLgam	0.036	0.036	0.036	0.032	0.037
	BLUiR	0.036	0.036	0.036	0.032	0.037
	LuceneScore (ours)	0.072	0.205	0.265	0.131	0.133
Teiid	ABLoTS (ours)	**0.312**	**0.582**	**0.691**	**0.372**	**0.429**
	AmaLgam	0.209	0.440	0.518	0.267	0.318
	BLUiR	0.206	0.429	0.500	0.252	0.307
	LuceneScore (ours)	0.230	0.496	0.599	0.307	0.356
Weld	ABLoTS (ours)	**0.288**	**0.518**	**0.613**	**0.343**	**0.394**
	AmaLgam	0.160	0.384	0.464	0.221	0.252
	BLUiR	0.144	0.376	0.464	0.207	0.237

	LuceneScore (ours)	0.288	0.440	0.528	0.328	0.368
Zookeeper	ABLoTS (ours)	**0.407**	**0.711**	**0.798**	**0.490**	**0.539**
	AmaLgam	0.283	0.630	0.728	0.412	0.433
	BLUiR	0.326	0.554	0.707	0.410	0.436
	LuceneScore (ours)	0.359	0.630	0.761	0.465	0.484

List of Tables

2.1. Important Issue Properties 18

3.1. Datasets and Projects used to Evaluate Approaches 33
3.2. Overview of IR-based Bug Localization Approaches 35

5.1. Tables in a Project's Database 52
5.2. Projects in SEOSS 55

6.1. ABLoTS Feature Vectors 71

7.1. Commit to Issue Linkage in SEOSS 75
7.2. Issue-to-Commit Linkage in SEOSS 75
7.3. Feature Matrix Encoding Issue-Commit Pairs 86

8.1. Key Figures for Projects in Curated Dataset 94
8.2. Confusion Matrix for a Binary Classifier 100
8.3. Studied Parameterization Variants of TraceScore 102

A.1. Comparison of Different Similar Issue Components IV
A.2. Comparison of Bug Localization Algorithms on Testing Data VI

List of Figures

2.1. Trace Link Directionality 14
2.2. Issue Overview of Project Derby 16
2.3. Example Issue of Project Derby 17
2.4. Issue Type Distribution in 33 Open-Source Projects 19
2.5. Issue Workflow Schema in Atlassian Jira 20
2.6. GitHub Screenshot of Project Derby 21
2.7. Example Commit in Project Derby 22
2.8. Tagged Commit Messages 23
2.9. Example for a Traceability Graph in Project Derby 24
2.10. Structure of IR-based Bug Localization 25
2.11. Preprocessing Steps for Lexical Analysis 27

3.1. Bug Localization Framework 37

4.1. Overview of the Holistic Approach 42
4.2. Artifact Model 44

5.1. Mining Process 50
5.2. Database Schema for a Project 54

6.1. Example Traceability Graph in Project Pig 58
6.2. Structure of TraceScore Component 62
6.3. Tracability Graph Example 64
6.4. Structure of LuceneScore 69
6.5. Structure of ABLoTS 71

7.1. Example of Failed Attempt to Create an Issue-to-Commit Trace Link 78
7.2. Artifact Relations During a Project's Lifetime 79
7.3. Late Commits 81
7.4. Source Code File Overlap 82

8.1. Datasets Used for Evaluation 93
8.2. Execution Schema of Experiment I 103
8.3. Comparision of Similar Issue Components (MAP, MRR) 104
8.4. Execution Schema of Experiment II 105
8.5. Comparison of Different TraceScore Parameterization (MAP, MRR) . 106

8.6. K-fold Cross-validation 108
8.7. Execution Schema of Experiment III for ABLoTS 109
8.8. Execution Schema of Experiment III for Other Algorithms 109
8.9. Comparison of Different Bug Localization Algorithms (MAP) 110
8.10. Execution Schema of Experiment IV 113
8.11. Trace Link Set Augmentation Performance (40% removal) 114
8.12. Execution Schema of Experiment V for ABLoTS 116
8.13. Execution Schema of Experiment V for Other Algorithms 116
8.14. Comparison of Different Bug Localization Algorithms (MAP) on Augmented Data (40%) 117

9.1. Comparison of Bug Localization Algorithm Components (MAP) . . . 124
9.2. Weighted Trace Link Set Augmentation Performance (40% removal) . 125
9.3. Weighted Trace Link Set Augmentation Performance (20% removal) . 127
9.4. Trace Link Set Augmentation Performance (100% removal) 128
9.5. Comparison of Similar Issue components (MAP) on Augmented Data (40% removal) 130
9.6. Traceability Graph Pruning 132
9.7. Traceability Graph Analysis 133

A.1. TraceScore History Length Comparision (MAP) II
A.2. TraceScore History Length Comparision (MRR) III

List of Abbreviations

ABLoTS	Automated Bug Localization using TraceScore
AmaLgam	Automated Localization of Bugs using Various Information (Wang and Lo 2014)
BLUiR	Bug Localization Using information Retrieval (Saha et al. 2013)
IR	Information retrieval
ITS	Issue Tracking System
CollabScore	Collaborative Filtering Score
JBT	Jboss-Transaction-Manager, project in SEOSS dataset
OSS	Open-source software
SEOSS	Software Engineering in Open-Source Systems
SimiScore	Similarity Score
TLSA	Trace Link Set Augmentation
VCS	Version Control System

References

Abreu, Rui, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. “A practical evaluation of spectrum-based fault localization.” *J. Syst. Softw.* 82 (11).

Agile Manifesto Team. 2001. “Manifesto for Agile Software Development.” Accessed September 4, 2021. https://agilemanifesto.org/.

Akbar, Shayan A., and Avinash C. Kak. 2019. “SCOR: source code retrieval with semantics and order.” In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada,* edited by Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc. IEEE. https://doi.org/10.1109/MSR.2019.00012.

———. 2020. “A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization.” In *MSR.* ACM.

Albon, Chris. 2018. *Machine learning with python cookbook: Practical solutions from preprocessing to deep learning.* ”O’Reilly Media, Inc.”.

Wang, Shaowei. 2017. “AmaLgam source code.” Accessed April 13, 2021.

Antoniol, Giuliano, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. “Recovering Traceability Links between Code and Documentation.” *IEEE Trans. Software Eng.* 28 (10).

Anvik, John, Lyndon Hiew, and Gail C Murphy. 2005. “Coping with an open bug repository.” In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange.*

———. 2006. “Who should fix this bug?” In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006,* edited by Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa. ACM. https://doi.org/10.1145/1134285.1134336.

Apache Lucene Developers. 2021. “Apache Lucene.” Accessed April 14, 2021.

Apache Software Foundation. 2020. “How Should I Apply Patches From A Contributor,” September 7, 2020. Accessed September 7, 2020. http://www.apache.org/dev/committers.html#applying-patches.

Asuncion, Hazeline U., Arthur U. Asuncion, and Richard N. Taylor. 2010. "Software traceability with topic modeling." In *ICSE (1).* ACM.

Atlassian Corporation. 2020a. "Process issues with smart commits." Accessed August 17, 2021. https://support.atlassian.com/jira-software-cloud/docs/process-issues-with-smart-commits/.

Bachmann, Adrian, and Abraham Bernstein. 2009. "Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects." In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops.* IWPSE-Evol '09. Amsterdam, The Netherlands: ACM. ISBN: 978-1-60558-678-6. https://doi.org/10.1145/1595808.1595830.

Bettenburg, Nicolas, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. "Extracting structural information from bug reports." In *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings,* edited by Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey. ACM. https://doi.org/10.1145/1370750.1370757.

Binkley, David, and Dawn Lawrie. 2010. "Information retrieval applications in software maintenance and evolution." *Encyclopedia of software engineering.*

Breiman, Leo. 2001. "Random forests." *Machine learning* 45 (1).

Breu, Silvia, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. "Information needs in bug reports: improving cooperation between developers and users." In *CSCW.* ACM.

Briand, Lionel C., Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, and Tao Yue. 2014. "Traceability and SysML design slices to support safety inspections: A controlled experiment." *ACM Trans. Softw. Eng. Methodol.* 23 (1). https://doi.org/10.1145/2559978.

Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, et al. 2013. "API design for machine learning software: experiences from the scikit-learn project." In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning,* 108–122.

Cambridge Judge Business School MBA, Undo IO. 2021. "The Business Value of Optimizing CI Pipelins." Accessed September 1, 2021. https://undo.io/the-cost-of-software-failures/.

Canfora, Gerardo, and Luigi Cerulo. 2005. "Impact Analysis by Mining Software and Change Request Repositories." In *11th IEEE International Symposium on Software Metrics (METRICS 2005), 19-22 September 2005, Como Italy.* IEEE Computer Society. https://doi.org/10.1109/METRICS.2005.28.

Center of Excellence for Software & Systems Traceability. 2020. "CoEST website." Accessed September 4, 2021. http://sarec.nd.edu/coest/index.html.

Cleland-Huang, Jane, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. 2007. "Best Practices for Automated Traceability." *Computer* 40 (6).

Cleland-Huang, Jane, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. "Software traceability: trends and future directions." In *FOSE.* ACM.

Cliff, Norman. 1993. "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin* 114 (3).

Committee, IEEE Standards Coordinating, et al. 1990. "IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos." *CA: IEEE Computer Society* 169:132.

Croft, W Bruce, Donald Metzler, and Trevor Strohman. 2010. *Search engines: Information retrieval in practice.* Vol. 520. Addison-Wesley Reading.

———. 2009. *Search Engines - Information Retrieval in Practice.* Pearson Education.

Cubranic, Davor, and Gail C. Murphy. 2003. "Hipikat: Recommending Pertinent Software Development Artifacts." In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA,* edited by Lori A. Clarke, Laurie Dillon, and Walter F. Tichy. IEEE Computer Society. https://doi.org/10.1109/ICSE.2003.1201219.

D'Haeseleer, Willem. 2019. "Jira Hot Linker." Accessed August 17, 2021. https://github.com/helmus/Jira-Hot-Linker.

Dallmeier, Valentin, and Thomas Zimmermann. 2007a. "Extraction of bug localization benchmarks from history." In *ASE.* ACM.

———. 2007b. "iBUGS Bug repositories extracted from project history." Accessed August 27, 2021. https://www.st.cs.uni-saarland.de/ibugs/.

De Lucia, Andrea, Fausto Fasano, and Rocco Oliveto. 2008. "Traceability management for impact analysis." In *2008 Frontiers of Software Maintenance.* IEEE.

Deerwester, Scott, Susan Dumais, Thomas Landauer, George Furnas, and Laura Beck. 1988. "Improving information-retrieval with latent semantic indexing." In *Proceedings of the ASIS annual meeting,* vol. 25.

Dekhtyar, Alex, Jane Huffman Hayes, Senthil Karthikeyan Sundaram, Elizabeth Ashlee Holbrook, and Olga Dekhtyar. 2007. “Technique Integration for Requirements Assessment.” In *RE.* IEEE Computer Society.

Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. “Feature location in source code: a taxonomy and survey.” *J. Softw. Evol. Process.* 25 (1).

Dumais, Susan T. 1991. “Improving the retrieval of information from external sources.” *Behavior research methods, instruments, & computers* 23 (2).

Eisenbarth, Thomas, Rainer Koschke, and Daniel Simon. 2003. “Locating Features in Source Code.” *IEEE Trans. Software Eng.* 29 (3).

Feldt, Robert. 2014. “Do System Test Cases Grow Old?” In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA.* IEEE Computer Society. https://doi.org/10.1109/ICST.2014.47.

Fischer, Michael, Martin Pinzger, and Harald C. Gall. 2003. “Analyzing and Relating Bug Report Data for Feature Tracking.” In *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003,* edited by Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey. IEEE Computer Society. https://doi.org/10.1109/WCRE.2003.1287240.

Flyvbjerg, Bent. 2006. “Five misunderstandings about case-study research.” *Qualitative inquiry* 12 (2).

Gay, Gregory, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. “On the use of relevance feedback in IR-based concept location.” In *ICSM.* IEEE Computer Society.

Gethers, Malcom, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. “On integrating orthogonal information retrieval methods to improve traceability recovery.” In *ICSM.* IEEE Computer Society.

Git Community. 2020. “Git.” Accessed August 14, 2021. https://git-scm.com.

Github, Inc. 2021. “Github.” Accessed August 16, 2021. https://github.com.

GitHub, Inc. 2018. “The State of the Octoverse 2018.” Accessed September 4, 2021. https://octoverse.github.com/.

———. 2019. “GitHub.com + Jira Software integration.” Accessed August 17, 2021. https://github.com/integrations/jira.

Goman, Maxim, Michael Rath, and Patrick Mäder. 2017. "Lessons Learned from Analyzing Requirements Traceability using a Graph Database." In *Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen,* 27–30. Gesellschaft für Informatik e.V.

Gonzalez, Danielle, Michael Rath, and Mehdi Mirakhorli. 2020. "Did You Remember To Test Your Tokens?" In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020.* ACM. https://doi.org/10.1145/3379597.3387471. https://doi.org/10.1145/3379597.3387471.

Gotel, O. C. Z., and Anthony Finkelstein. 1994. "An analysis of the requirements traceability problem." In *ICRE.* IEEE Computer Society.

Gotel, Orlena, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan I. Maletic, and Patrick Mäder. 2012. "Traceability Fundamentals." In *Software and Systems Traceability.* Springer.

Grissom, Robert J, and John J Kim. 2012. *Effect sizes for research: Univariate and multivariate applications.* Routledge.

Guo, Jin, Jinghui Cheng, and Jane Cleland-Huang. 2017. "Semantically enhanced software traceability using deep learning techniques." In *ICSE.* IEEE.

Guo, Lan, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. "Robust Prediction of Fault-Proneness by Random Forests." In *Proceedings of the 15th International Symposium on Software Reliability Engineering,* 417–428. ISSRE '04. Washington, DC, USA: IEEE Computer Society. ISBN: 0-7695-2215-7. https://doi.org/10.1109/ISSRE.2004.35.

Hadoop Community. 2020. "How to Commit." Accessed September 4, 2021. https://cwiki.apache.org/confluence/display/HADOOP2/HowToCommit.

Hassan, Ahmed E. 2008. "The road ahead for mining software repositories." In *2008 Frontiers of Software Maintenance.* IEEE.

Hayes, Jane Huffman, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods." *IEEE Trans. Software Eng.* 32 (1).

Heindl, Matthias, and Stefan Biffl. 2005. "A case study on value-based requirements tracing." In *ESEC/SIGSOFT FSE.* ACM.

Herzig, Kim, Sascha Just, and Andreas Zeller. 2013. “It’s not a bug, it’s a feature: how misclassification impacts bug prediction.” In *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013.* IEEE Computer Society. https://doi.org/10.1109/ICSE.2013.6606585.

Herzig, Kim, and Andreas Zeller. 2014. “Mining Bug Data - A Practitioner’s Guide.” In *Recommendation Systems in Software Engineering,* edited by Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. Springer. https://doi.org/10.1007/978-3-642-45135-5_6.

Highsmith, Jim, and Alistair Cockburn. 2001. “Agile software development: The business of innovation.” *Computer* 34 (9).

Hinsen, Konrad, Konstantin Läufer, and George K. Thiruvathukal. 2009. “Essential Tools: Version Control Systems.” *Comput. Sci. Eng.* 11 (6).

Holtmann, Jörg, Jan-Philipp Steghöfer, Michael Rath, and David Schmelter. 2020. “Cutting through the Jungle: Disambiguating Model-based Traceability Terminology.” In *RE,* XXX–YYY. IEEE.

Hovemeyer, David, and William Pugh. 2004. “Finding bugs is easy.” *Acm sigplan notices* 39 (12).

IEEE Spectrum. 2017. “The 2017 top programming languages.” Accessed September 4, 2020. https://spectrum.ieee.org/computing/software/the-2017-top-programmin g-languages.

International Electronical Commission. 2003. *IEC 61511-1 ed 1.0, Safety Instrumented Systems for the Process Industry Sector.*

International Organization for Standardization. 2011. *ISO 26262:1:2011 Road Vehicles - Functional Safety.*

JavaParser Developers. 2021. “JavaParser.” Accessed April 14, 2021.

Atlassian Corporation. 2020b. “Jira website.” Accessed August 14, 2021. https://www.atlassian.com/software/jira.

———. 2021a. “Issue properties.” Accessed August 16, 2021. https://support.atlassian.com/jira-cloud-administration/docs/what-are-issue-statuses-priorities-and-resolutions/.

———. 2021b. “What are issue types.” Accessed August 16, 2021. https://support.atlassian.com/jira-cloud-administration/docs/what-are-issue-types/.

Jones, James A., and Mary Jean Harrold. 2005. “Empirical evaluation of the tarantula automatic fault-localization technique.” In *ASE.* ACM.

Kagdi, Huzefa H., Michael L. Collard, and Jonathan I. Maletic. 2007. "A survey and taxonomy of approaches for mining software repositories in the context of software evolution." *J. Softw. Maintenance Res. Pract.* 19 (2).

Kamei, Yasutaka, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. "A Large-Scale Empirical Study of Just-in-Time Quality Assurance." *IEEE Trans. Software Eng.* 39 (6). https://doi.org/10.1109/TSE.2012.70.

Kaur, Arvinder, and Vidhi Vig. 2016. "Challenges in data extraction from Open Source software repositories." In *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence).* IEEE.

Keenan, Ed, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, et al. 2012. "TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions." In *ICSE.* IEEE Computer Society.

Kim, Dongsun, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. "Where Should We Fix This Bug? A Two-Phase Recommendation Model." *IEEE Trans. Software Eng.* 39 (11).

Kim, Sunghun, E. James Whitehead Jr., and Yi Zhang. 2008. "Classifying Software Changes: Clean or Buggy?" *IEEE Trans. Software Eng.* 34 (2).

Kim, Sunghun, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. "Predicting Faults from Cached History." In *ICSE.* IEEE Computer Society.

Knethen, Antje von, Barbara Paech, Friedemann Kiedaisch, and Frank Houdek. 2002. "Systematic Requirements Recycling through Abstraction and Traceability." In *RE.* IEEE Computer Society.

Koch, Stefan. 2004. "Agile principles and open source software development: A theoretical and empirical discussion." In *International Conference on Extreme Programming and Agile Processes in Software Engineering.* Springer.

Kochhar, Pavneet Singh, Yuan Tian, and David Lo. 2014. "Potential biases in bug localization: do they matter?" In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.* ACM. https://doi.org/10.1145/2642937.2642997.

Kuang, Hongyu, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. 2015. "Can method data dependencies support the assessment of traceability between requirements and source code?" *J. Softw. Evol. Process.* 27 (11).

Kuang, Hongyu, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lv, and Alexander Egyed. 2012. "Do data dependencies in source code complement call dependencies for understanding requirements traceability?" In *ICSM.* IEEE Computer Society.

Kuang, Hongyu, Jia Nie, Hao Hu, Patrick Rempel, Jian Lu, Alexander Egyed, and Patrick Mäder. 2017. "Analyzing closeness of code dependencies for improving IR-based Traceability Recovery." In *SANER.* IEEE Computer Society.

Lam, An Ngoc, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. "Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)." In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015,* edited by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society. https://doi.org/10.1109/ASE.2015.73.

Le, Tien-Duy B., Mario Linares Vasquez, David Lo, and Denys Poshyvanyk. 2015. "RCLinker: automated linking of issue reports and commits leveraging rich contextual information." In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015,* edited by Andrea De Lucia, Christian Bird, and Rocco Oliveto. IEEE Computer Society. https://doi.org/10.1109/ICPC.2015.13.

Lewis, C, and R Ou. 2011. "Bug prediction at google." Accessed September 4, 2021. http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html.

Liebchen, Gernot Armin, and Martin J. Shepperd. 2016. "Data Sets and Data Quality in Software Engineering: Eight Years On." In *PROMISE.* ACM.

Liu, Chao, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. 2006. "Statistical Debugging: A Hypothesis Testing-Based Approach." *IEEE Trans. Software Eng.* 32 (10).

Lohar, Sugandha, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. "Improving trace accuracy through data-driven configuration and composition of tracing features." In *ESEC/SIGSOFT FSE.* ACM.

Lucia, Andrea De, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2004. "Enhancing an Artefact Management System with Traceability Recovery Features." In *ICSM.* IEEE Computer Society.

Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. "Are faults localizable?" In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland,* edited by Michele Lanza, Massimiliano Di Penta, and Tao Xie. IEEE Computer Society. https://doi.org/10.1109/MSR.2012.6224302.

Lukins, Stacy K., Nicholas A. Kraft, and Letha H. Etzkorn. 2008. “Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation.” In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008,* edited by Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta. IEEE Computer Society. https://doi.org/10.1109/WCRE.2008.33.

Mäder, Patrick, and Jane Cleland-Huang. 2015. “From Raw Project Data to Business Intelligence.” *IEEE Software* 32 (4).

Mäder, Patrick, and Alexander Egyed. 2011. “Do software engineers benefit from source code navigation with traceability? - An experiment in software change management.” In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011,* edited by Perry Alexander, Corina S. Pasareanu, and John G. Hosking. IEEE Computer Society. https://doi.org/10.1109/ASE.2011.6100095.

Mahmoud, Anas, and Nan Niu. 2010. “Using Semantics-Enabled Information Retrieval in Requirements Tracing: An Ongoing Experimental Investigation.” In *COMPSAC.* IEEE Computer Society.

———. 2011. “TraCter: A tool for candidate traceability link clustering.” In *RE.* IEEE Computer Society.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* New York: Cambridge University Press. ISBN: 978-0-521-86571-5.

Manning, Christopher D., and Hinrich Schütze. 2001. *Foundations of statistical natural language processing.* MIT Press.

Marcus, Andrian, and Jonathan I. Maletic. 2003. “Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing.” In *ICSE.* IEEE Computer Society.

McMillan, Collin, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. “Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications.” *IEEE Trans. Software Eng.* 38 (5).

Merten, Thorsten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech. 2016. “Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data?” In *REFSQ,* vol. 9619. Lecture Notes in Computer Science. Springer.

Mockus, Audris, Roy T Fielding, and James Herbsleb. 2000. “A case study of open source software development: the Apache server.” In *Proceedings of the 22nd international conference on Software engineering.*

Moreno, Laura, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. "On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization." In *ICSME.* IEEE Computer Society.

Murgia, Alessandro, Parastou Tourani, Bram Adams, and Marco Ortu. 2014. "Do developers feel emotions? an exploratory analysis of emotions in software artifacts." In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India,* edited by Premkumar T. Devanbu, Sung Kim, and Martin Pinzger. ACM. https://doi.org/10.1145/2597073.2597086.

Nguyen, Anh Tuan, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. "A topic-based approach for narrowing the search space of buggy files from a bug report." In *ASE.* IEEE Computer Society.

Niu, Nan, Tanmay Bhowmik, Hui Liu, and Zhendong Niu. 2014. "Traceability-enabled refactoring for managing just-in-time requirements." In *RE.* IEEE Computer Society.

Olson, David L, and Dursun Delen. 2008. *Advanced data mining techniques.* Springer Science & Business Media.

Panichella, Annibale, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2013. "When and How Using Structural Information to Improve IR-Based Traceability Recovery." In *CSMR.* IEEE Computer Society.

Porter, Martin. 2006. *Porter Stemmer website.* http://tartarus.org/~martin/PorterStemmer/.

Poshyvanyk, Denys, Andrian Marcus, Vaclav Rajlich, Yann-Gaël Gueheneuc, and Giuliano Antoniol. 2006. "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification." In *ICPC.* IEEE Computer Society.

Project Management Institute. 2015. "Pulse of the Profession 2015." Accessed September 4, 2021. https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2015.pdf.

Project Management Zone. 2018. "Project Management Systems - Popularity Ranking." Accessed August 26, 2021. https://project-management.zone/ranking/issue.

Rao, Shivani, and Avinash C. Kak. 2011. "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models." In *MSR.* ACM.

Rath, Michael, David Akehurst, Christoph Borowski, and Patrick Mäder. 2017. "Are Graph Query Languages Applicable for Requirements Traceability Analysis?" In *REFSQ Workshops,* vol. 1796. CEUR Workshop Proceedings. CEUR-WS.org.

Rath, Michael, Maksim Goman, and Patrick Mäder. 2017. "State of the Art of Traceability in Open-Source Projects." In *Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen,* 8–11. Gesellschaft für Informatik e.V.

Rath, Michael, David Lo, and Patrick Mäder. 2018. "Analyzing requirements and traceability information to improve bug localization." In *MSR,* 442–453. ACM.

Rath, Michael, and Patrick Mäder. 2018. "Influence of Structured Information in Bug Report Descriptions on IR-Based Bug Localization." In *SEAA,* 26–32. IEEE Computer Society.

———. 2019a. "Structured information in bug report descriptions - influence on IR-based bug localization and developers." *Software Quality Journal* 27 (3): 1315–1337.

———. 2019b. "The SEOSS 33 dataset - Requirements, bug reports, code history, and trace links for entire projects." *Data in brief* 25:104005.

———. 2020. "Request for Comments: Conversation Patterns in Issue Tracking Systems of Open-Source Projects." In *SAC.* ACM.

Rath, Michael, Patrick Rempel, and Patrick Mäder. 2017. "The IlmSeven Dataset." In *RE,* 516–519. IEEE Computer Society.

Rath, Michael, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. "Traceability in the wild: automatically augmenting incomplete trace links." In *ICSE,* 834–845. ACM.

Rath, Michael, Mihaela Todorova Tomova, and Patrick Mäder. 2019. "Selecting Open Source Projects for Traceability Case Studies." In *REFSQ,* 11412:229–242. Lecture Notes in Computer Science. Springer.

———. 2020. "SpojitR: Intelligently Link Development Artifacts." In *SANER.* IEEE.

Regnell, Björn, Richard Berntsson-Svensson, and Krzysztof Wnuk. 2008. "Can We Beat the Complexity of Very Large-Scale Requirements Engineering?" In *Requirements Engineering: Foundation for Software Quality, 14th International Working Conference, REFSQ 2008, Montpellier, France, June 16-17, 2008, Proceedings,* edited by Barbara Paech and Colette Rolland, vol. 5025. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-540-69062-7_11.

Rempel, Patrick, and Patrick Mäder. 2017. "Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality." *IEEE Trans. Software Eng.* 43 (8). https://doi.org/10.1109/TSE.2016.2622264.

Rempel, Patrick, Patrick Mäder, and Tobias Kuschke. 2013. "Towards feature-aware retrieval of refinement traces." In *TEFSE@ICSE.* IEEE Computer Society.

Ripon Saha. 2016. "BLUiR-Package." Accessed April 14, 2021.

Robertson, Stephen E., Steve Walker, and Micheline Beaulieu. 2000. "Experimentation as a way of life: Okapi at TREC." *Information processing & management* 36 (1).

Romo, Bilyaminu Auwal, Andrea Capiluppi, and Tracy Hall. 2014. "Filling the Gaps of Development Logs and Bug Issue Data." In *Proceedings of The International Symposium on Open Collaboration, OpenSym 2014, Berlin, Germany, August 27 - 29, 2014.* ACM. https://doi.org/10.1145/2641580.2641592.

Ruan, Hang, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2019. "DeepLink: Recovering issue-commit links based on deep learning." *J. Syst. Softw.* 158. https://doi.org/10.1016/j.jss.2019.110406.

Runeson, Per, and Martin Höst. 2009. "Guidelines for conducting and reporting case study research in software engineering." *Empir. Softw. Eng.* 14 (2). https://doi.org/10.1007/s10664-008-9102-8.

Saha, Ripon K., Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. "Improving bug localization using structured information retrieval." In *28th IEEE/ACM Int. Conference on Automated Software Engineering, ASE 2013.* https://doi.org/10.1109/ASE.2013.6693093. https://doi.org/10.1109/ASE.2013.6693093.

Schermann, Gerald, Martin Brandtner, Sebastiano Panichella, Philipp Leitner, and Harald C. Gall. 2015. "Discovering loners and phantoms in commit and issue data." In *ICPC.* IEEE Computer Society.

Schröter, Adrian, Nicolas Bettenburg, and Rahul Premraj. 2010. "Do stack traces help developers fix bugs?" In *MSR.* IEEE Computer Society.

Si, XiaoSheng, ChangHua Hu, and ZhiJie Zhou. 2010. "Fault prediction model based on evidential reasoning approach." *Science China Information Sciences* 53 (10): 2032–2046.

Singh, VB, and Krishna Kumar Chaturvedi. 2011. "Bug tracking and reliability assessment system (btras)." *International Journal of Software Engineering and Its Applications* 5 (4).

Skerrett, Ian. 2011. *The Eclipse Foundation (2011). The Eclipse Community Survey 2011.* Technical report. Tech. rep. Ottawa, Ontario, Canada: The Eclipse Foundation (cit. on pp. 3, 19).

Sklearn Documentation. 2021. "Cross-validation." Accessed July 24, 2021. https://scikit-learn.org/stable/modules/cross_validation.html.

Sliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller. 2005. "When do changes induce fixes?" In *MSR.* ACM.

Stackoverflow. 2018. "Developer Survey Results 2018." Accessed August 26, 2021. https://insights.stackoverflow.com/survey/2018#overview.

Strohman, Trevor, Donald Metzler, Howard Turtle, and W Bruce Croft. 2005. "Indri: A language model-based search engine for complex queries." In *Proceedings of the international conference on intelligent analysis,* vol. 2. 6. Citeseer.

Subversion Community. 2021. "Coding and Commit Conventions." Accessed September 4, 2021. https://subversion.apache.org/docs/community-guide/conventions.html#log-messages.

Sullivan, Matthew. 2014. "Linkify Jira Issues." Accessed August 17, 2021. https://bit.ly/2K9I6IS.

Sultanov, Hakim, Jane Huffman Hayes, and Wei-Keat Kong. 2011. "Application of swarm techniques to requirements tracing." *Requir. Eng.* 16 (3).

Sun, Yan, Qing Wang, and Ye Yang. 2017. "FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance." *Inf. Softw. Technol.,* https://doi.org/10.1016/j.infsof.2016.11.010.

The Lemur Project. 2020. "Indri Toolkit." Accessed April 14, 2021.

TIOBE Software BV. 2021. "TIOBE Index." Accessed August 26, 2021. https://www.tiobe.com/tiobe-index/.

Tomova, Mihaela Todorova, Michael Rath, and Patrick Mäder. 2017. "Preprocessing Texts in Issue Tracking Systems to improve IR Techniques for Trace Creation." In *Workshop des Arbeitskreises Traceability/Evolution der Technischen Universität Ilmenau: Aktuelle Methoden zur Gewinnung und Aktualisierung von Traceability-Modellen,* 17–20. Gesellschaft für Informatik e.V.

———. 2018. "Use of trace link types in issue tracking systems." In *ICSE (Companion Volume),* 181–182. ACM.

Voorhees, Ellen M. 1999. "The TREC-8 Question Answering Track Report." In *TREC,* vol. 500-246. NIST Special Publication. National Institute of Standards / Technology (NIST).

Wang, Qianqian, Chris Parnin, and Alessandro Orso. 2015. "Evaluating the usefulness of IR-based fault localization techniques." In *ISSTA.* ACM.

Wang, Shaowei, and David Lo. 2014. "Version history, similar report, and structure: putting them together for improved bug localization." In *22nd International Conference on Program Comprehension, ICPC 2014.* https://doi.org/10.1145/2597008.2597148.

———. 2016. "AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization." *Journal of Software: Evolution and Process* 28 (10). https://doi.org/10.1002/smr.1801.

Wen, Ming, Rongxin Wu, and Shing-Chi Cheung. 2016. "Locus: locating bugs from software changes." In *ASE.* ACM.

Wilcoxon, Frank. 1992. "Individual comparisons by ranking methods." In *Breakthroughs in statistics.* Springer.

Wong, Chu-Pan, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis." In *IEEE International Conference on Software Maintenance and Evolution.* https://doi.org/10.1109/ICSME.2014.40.

Wong, W. Eric, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. "A Survey on Software Fault Localization." *IEEE Trans. Software Eng.* 42.

Xie, Tao, Lu Zhang, Xusheng Xiao, Ying-Fei Xiong, and Dan Hao. 2014. "Cooperative software testing and analysis: Advances and challenges." *Journal of Computer Science and Technology* 29 (4): 713–723.

Ye, Xin, Razvan C. Bunescu, and Chang Liu. 2014. "Learning to rank relevant files for bug reports using domain knowledge." In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014,* edited by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM. https://doi.org/10.1145/2635868.2635874.

Ye, Xin, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. "From word embeddings to document similarities for improved information retrieval in software engineering." In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016,* edited by Laura K. Dillon, Willem Visser, and Laurie Williams. ACM. https://doi.org/10.1145/2884781.2884862.

Yin, Robert K. 2009. *Case study research: design and methods.* 4th. Applied social research methods. Sage Publications. ISBN: 978-1-4129-6099-1.

Zeller, Andreas. 2009. *Why programs fail: a guide to systematic debugging.* Elsevier.

Zhang, Jie, Xiaoyin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. 2015. "A survey on bug-report analysis." *Sci. China Inf. Sci.* 58 (2). https://doi.org/10.1007/s11432-014-5241-2.

Zhao, Fei, Yaming Tang, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2015. "Is Learning-to-Rank Cost-Effective in Recommending Relevant Files for Bug Localization?" In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015.* IEEE. https://doi.org/10.1109/QRS.2015.49.

Zhou, Jian, Hongyu Zhang, and David Lo. 2012. "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports." In *ICSE.* IEEE Computer Society.

Zimmermann, Thomas, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. "What Makes a Good Bug Report?" *IEEE Trans. Software Eng.* 36 (5).

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß § 7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Erfurt, 25 November 2021

Michael Rath