University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

MIKOŁAJ FEJZER

MINING SOFTWARE REPOSITORIES FOR CODE QUALITY

PhD dissertation

SUPERVISOR:

prof. dr hab. Krzysztof Stencel
Institute of Informatics,
University of Warsaw

AUXILIARY SUPERVISOR:

dr Piotr Przymus
Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University

Toruń, April 2020

Author's declaration: aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

Mikołaj Fejzer,
27th April 2020

Supervisor's declaration: the dissertation is ready to be reviewed.

prof. dr hab. Krzysztof Stencel,
27th April 2020

Auxiliary Supervisor's declaration: the dissertation is ready to be reviewed.

dr Piotr Przymus,
27th April 2020

ABSTRACT

This dissertation covers a series of code quality topics utilizing mining of software repositories. We focus on code review support and software bugs detection. The code review is proof-reading of proposed code change, accepted as an industry standard. The reviewers find common shortcomings such as lacking test coverage, misused design patterns, or logic errors and provide feedback to the author of changes. Quality of review depends on correct selection of reviewers. Most software is shipped with various kinds of defects. Fixing those defects is one of most common activities in software development. Bug localization is a process of finding specific defects in project source code, based on user supplied reports.

First, we analyze open source projects to gather insights on contributor activities and bug prevalence, with the goal of helping bug detection during code review. We use topic modeling on change comments to elect core developers to be involved in the code review. Additionally, we examine both commit comments and issue topics per each month to assess the popularity of bug fixing. Consequently, due to the fact that over half of all comments are related to bugs, we focus on improving code review by introducing change classifier. The classifier indicates potentially buggy changes during code review.

Second, we present a new method of recommending code reviewers, utilizing profiles of individual contributors. For each developer we maintain a corresponding profile, based on a multiset of all file path segments from commits reviewed by him/her. The profile is updated after participation in the new review. We employ a similarity function between such profiles and change proposals to be reviewed. The contributor whose profile is the most similar becomes the recommended reviewer. We performed an experimental comparison of our method against state-of-the-art techniques using four large open-source projects. We obtained improved results in terms of classification metrics (precision, recall and F-measure) and performance (we have lower time and space complexity).

Third, we propose adaptive method to localize bugs based on bug reports. Upon receiving a new bug report, developers need to find its cause in the source code. Bug localization can be supported by a tool that ranks all source files according to how likely they include the bug. Consequently, we introduce new feature weighting approaches and an adaptive selection algorithm. We evaluate localization method on publicly available datasets, with competitive results and performance compared to state–of–the–art.

KEYWORDS: Bug localization, Code review, Mining software repositories, Reviewer recommendation

ACM COMPUTING CLASSIFICATION: Computing methodologies~Feature selection, Computing methodologies~Learning to rank, Information systems~Clustering and classification, Information systems~Recommender systems, Information systems~Structured text search, Software and its engineering~Collaboration in software development, Software and its engineering~Software defect analysis

STRESZCZENIE

Niniejsza rozprawa obejmuje szereg zagadnień związanych z jakością kodu bazując na eksploracji repozytoriów. Koncentrujemy się na wsparciu inspekcji kodu i wykrywaniu błędów programistycznych. Inspekcja kodu jest techniką standardowo stosowaną w przemyśle. Polega na badaniu zaproponowanych zmian w kodzie źródłowym przez innych programistów. Recenzenci znajdują typowe niedociągnięcia i przekazują informacje zwrotne autorowi zmian. Jakość recenzji zależy od właściwego doboru recenzentów. Większość oprogramowania jest dostarczana z różnego rodzaju defektami. Naprawa tych wad jest jednym z najczęstszych działań w tworzeniu oprogramowania. Lokalizacja błędów to proces znajdowania określonych defektów w kodzie źródłowym projektu na podstawie raportów dostarczonych przez użytkownika.

Pierwszym rozważanym zagadnieniem jest analiza projektów open source, aby zaobserwować zachowanie kontrybutorów i występowanie błędów. Używamy modelowania tematów na treści komentarzy do zestawów zmian, żeby znaleźć głównych programistów dla każdego badanego projektu. Ponadto badamy tematy zgłoszeń użytkowników jak i zestawów zmian, aby ocenić popularność naprawiania błędów w skali miesięcy. Na podstawie powyższych analiz skupiamy się na ulepszeniu inspekcji kodu poprzez dodanie klasyfikatora zmian, ponieważ ponad połowa wszystkich komentarzy dotyczy błędów. Klasyfikator wskazuje potencjalnie błędne zmiany podczas inspekcji.

Jako drugie poruszane zagadnienie przedstawiamy nową metodę rekomendacji recenzentów kodu wykorzystującą profile poszczególnych kontrybutorów. Dla każdego programisty utrzymujemy odpowiedni profil oparty o wielozbiór wszystkich segmentów ścieżek recenzowanych dotychczas plików. Profil jest aktualizowany po przygotowaniu nowej recenzji. Stosujemy funkcję podobieństwa między takimi profilami i propozycjami zmian wymagających inspekcji. Kontrybutor, którego profil jest najbardziej podobny, zostaje rekomendowanym recenzentem. Przeprowadziliśmy eksperymenty w celu porównania naszej metody z najnowocześniejszymi technikami, wykorzystując cztery duże projekty open source. Uzyskaliśmy lepsze wyniki pod względem miar jakości klasyfikacji oraz wydajności.

Trzecim zagadnieniem jest nowa, adaptacyjna metoda lokalizacji błędów na podstawie zgłaszanych raportów. Po otrzymaniu nowego raportu programiści muszą znaleźć przyczynę błędu w kodzie źródłowym. Proponujemy narzędzie wspierające lokalizację błędów za pomocą określenia prawdopodobieństwa zawierania usterki przez pliki w projekcie. Nasza metoda bazuje na nowych sposobach ważenia cech i adaptacyjnych algorytmach selekcji. Uzyskaliśmy konkurencyjne wyniki i wydajność w porównaniu do najnowocześniejszych technik na publicznie dostępnych zestawach danych.

SŁOWA KLUCZOWE: Lokalizacja błędów, Inspekcja kodu źródłowego, Eksploaracja repozytoriów oprogramowania, Rekomendowanie recenzentów

ACKNOWLEDGEMENTS

I want to thank my supervisors, dr Piotr Przymus and prof. dr hab. Krzysztof Stencel, for their encouragement, guidance, patience and support without which this thesis would not have been possible.
I would also like to thank all my other co-authors: dr Jakub Narębski, dr Marta Burzańska, dr hab. Piotr Wiśniewski and Michał Wojtyna, for all I have learned while working with them.

Finally, I wish to express my gratitude to my family for their support and patience.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Problem statement 2
- 1.3 Dissertation outline 3

2 PRELIMINARIES 5
- 2.1 Software engineering 5
- 2.2 Information retrieval 7
- 2.3 Learning 8
- 2.4 Mining software repositories 11

3 PRELIMINARY STUDIES ON OPEN SOURCE DATA 15
- 3.1 Introduction 15
- 3.2 Related work 15
- 3.3 Contribution analysis 18
- 3.4 Bug detection based on commit similarity 20
- 3.5 Concluding Remarks 22

4 RECOMMENDATION OF CODE REVIEWERS 25
- 4.1 Introduction 25
- 4.2 Related work 25
- 4.3 Problem statement 29
- 4.4 The proposed method 30
- 4.5 Evaluation Results 35
- 4.6 Discussion 43
- 4.7 Concluding Remarks 44

5 ADAPTIVE BUG LOCALIZATION BASED ON BUG REPORTS 45
- 5.1 Introduction 45
- 5.2 Related work 46
- 5.3 Problem statement 49
- 5.4 Feature engineering 49
- 5.5 Proposed solution 51
- 5.6 Evaluation Results 56
- 5.7 Discussion 60
- 5.8 Concluding Remarks 65

6 CONCLUSIONS 67
- 6.1 Future work 67

7 APPENDIX: REPLICATION AND DATASETS 69
- 7.1 Replication repository 69
- 7.2 Preliminary studies datasets 69

BIBLIOGRAPHY 75

LIST OF FIGURES

Figure 3.1 History limit tests 23
Figure 4.1 Evaluated projects insights 38
Figure 4.2 Metrics comparison 39
Figure 4.3 Recommended reviewers in Top-1 Tversky No Ext without tie-breaking 41
Figure 4.4 Performance difference between methods 43
Figure 5.1 Bug localization improvement graph 47
Figure 5.2 The block schema of a general learning to rank approach in bug localization. 52
Figure 5.3 Fine grained dataset [165] Accuracy@k results 58
Figure 5.4 Adaptive scoring MAP for different scoring functions 63
Figure 5.5 Adaptive Regression regularization 63
Figure 5.6 Adaptive Regression MAP Distribution 64

LIST OF TABLES

Table 3.1 Statistics of commit commenter groups 19
Table 3.2 Core contributors 19
Table 3.3 Preliminary tests 22
Table 3.4 History length groups 22
Table 3.5 History length tests 23
Table 4.1 Example repository and assignment of reviewers. 31
Table 4.2 Profile model operations 33
Table 4.3 Profile model modified operations 34
Table 4.4 Statistics of processed projects 36
Table 4.5 The recall and MRR for all methods and four projects 40
Table 4.6 The recall and MRR of Tversky No Ext without tie-breaking. 41
Table 4.7 Results of the performance evaluation 42
Table 4.8 Memory footprint of Tversky No Ext (in MB) 43
Table 5.1 Bug localization related work 46
Table 5.2 Used features 50
Table 5.3 Learning to rank bug localization methods comparison 53
Table 5.4 Scoring functions and weight schemata 54
Table 5.5 Datasets used in this chapter 56
Table 5.6 Fine grained dataset [165] MAP and MRR results 59
Table 5.7 BugLocator dataset [172] results 60
Table 5.8 Adaptive method performance on Ye et al. dataset [165] 61
Table 7.1 Contributors 70
Table 7.2 Bug prevalence 71
Table 7.3 Bug detection - small repositories 72
Table 7.4 Bug detection - medium repositories 72
Table 7.5 Bug detection - large repositories 73
Table 7.6 Bug detection - XL repositories 73

1 INTRODUCTION

Contents

1.1 Motivation 1
1.2 Problem statement 2
1.3 Dissertation outline 3

1.1 MOTIVATION

Mining of Software Repositories is a subfield of Software Engineering that studies and develops instruments useful for analyzing the rich data that are produced during software evolution [62]. Evolution of software was studied extensively [81, 157], with various models applied to quality measurement [94] or code decay detection [31]. Only relatively recent advances in information retrieval and machine learning enabled large scale data mining and knowledge extraction in this area [15, 47, 48, 64]. Consequently, new research trends arose, with the goal of software quality improvement via applying MSR [62, 124]. The most crucial applications of MSR in terms of software quality are bug prevention and localization [25, 67, 119, 139]. Among other important usages we may list code review enhancement [96], contribution detection and team dynamics analysis, with a focus on developer interactions and the social side of programming [130]. The software architecture is also studied via code reuse pattern discovery, coupling detection, architecture management and refactorisation support [47].

The data analyzed by MSR is stored in version control systems, bug/issue trackers, project management software, communication archives and code review systems [62]. Intuitively, we can divide available data sources into two groups: user supplied unstructured text and source code artifacts, with the former obtained from specific revision from project version control repository [1] and the later from other sources. Both groups are usually cross-linked, to represent coocuring changes of different project artifacts [63]. Examples of such links are issue tracker ticket identification numbers present in commit metadata message, repository tags joining branches to specific release, code comments referencing external sources and developers emails containing commit references. Depending on scope of research question and applied methodology either each commit is investigated or various commits are aggregated for further processing as specific versions of analyzed software [62]. Advent of decentralized version control systems, such as git, introduced unique advantages over centralized systems, for example metadata locality, possibility of repository conversion with intact history and improvements to line content tracking [15]. Consequently, hosting sites like GitHub or Bitbucket provide a valuable and complete source of data on many open source projects and their contributors [64].

In this dissertation we focus on ensuring software quality via applying MSR to various aspects of modern software development. In particular, we propose methods to select ap-

propriate code reviewers, detect new bugs similar to already solved ones and localize bugs based on user submitted reports.

The practice of peer code reviews is one of standards both within industry and open source communities. Thanks to code review, new contributors to the project are able to learn the domain quicker and receive constant feedback about their progress [108]. Other advantages of the code review process are better adjusted test coverage and lower number of bugs per each release [96]. Additionally, it enforces consistent adherence to both language standards and specific project guidelines of new code submissions [108]. Nevertheless, code review can be a time consuming activity, requiring cooperation and mutual understanding between code author and reviewer. To do the review properly potential reviewer candidates need to know specific project areas related to the changes. Long time developers tend to "take ownership" of specific modules or features, and maintain them [101], which includes review participation. The reviews without pre-assigned reviewers tend to either be abandoned or wait longer than others for any kind of activity [29, 57]. Furthermore incorrect assignment (beyond expertise of the reviewer) might cause the review to suffer from Parkinson's law of triviality. According to it, such reviewers would focus on unimportant details [124]. This demonstrates that finding competent reviewers quickly is an important problem. Consequently, we focus on the automation of the code reviewer recommendation. Before a change is merged into the destination branch within the repository, it is usually checked via build automation tools. These tools evaluate if changes pass compilation, unit and integration tests. Additional tools based on static code analysis are optionally included to provide more insight for reviewers. We propose another source of insight for reviewers, based on the history of already fixed bugs, suggesting if changed files are similar to those requiring fixes in the past.

Most of the software projects are delivered containing bugs, with more than one third of software development costs being spent on bug removal [39, 170]. Software users submit bug reports to bug trackers, such reports are one of the main sources of information about bugs [62]. Each report is managed by project maintainers, who triage, prioritize, remove duplicated issues, conduct replication and resolve bugs. The bug localization is the process of finding appropriate, defective source files within the project. Due to varying quality of bug reports, omitted data, nonstandard conditions occurring in the program and size of the project, the selection of relevant files may be a non-trivial task. Moreover, the developer must understand the architecture of the program and the domain of specific use cases to localize the correct source code successfully. We introduce adaptive method for automated bug localization, with the goal of decreasing time spent by developers on bug fixing activity, and consequently reducing costs of software creation and maintenance.

1.2 PROBLEM STATEMENT

The subject of this PhD dissertation is to improve the overall code quality using defect prevention and localization utilizing information retrieval and machine learning. The main objectives were the following:

- to recommend the code reviewers [34]
- to localize buggy files based on user report [33]

Additionally, we include preliminary studies on open source projects, gathering insights on contributor roles and bug detection [35, 36].

1.3 DISSERTATION OUTLINE

The dissertation consists of six main chapters and appendix.

- In Chapter 2 we introduce information retrieval and machine learning methods we use throughout the dissertation, along with software engineering datasources.
- In Chapter 3 we present groundwork studies [35, 36].
- In Chapter 4 we describe code reviewer recommendation algorithm [34].
- In Chapter 5 we introduce adaptive bug localization algorithm based on bug reports [33].
- In Chapter 6 we summarize our main findings and highlight some possible future work.
- In Appendix Chapter 7 we present replication repository and additional details of selected datasets used in this dissertation.

REFERENCES

[33] Mikolaj Fejzer, Jakub Narebski, Piotr Przymus and Krzysztof Stencel. "Tracking Buggy Files: New Efficient Adaptive Bug Localization Method". 2020. Manuscript.

[34] Mikolaj Fejzer, Piotr Przymus and Krzysztof Stencel. "Profile based recommendation of code reviewers". In: *Journal of Intelligent Information Systems* 50.3 (2018), pp. 597–619. DOI: 10.1007/s10844-017-0484-1.

[35] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski and Krzysztof Stencel. "Open Source Is a Continual Bugfixing by a Few". In: *Advances in Databases and Information Systems - 18th East European Conference, ADBIS 2014, Ohrid, Macedonia, September 7-10, 2014. Proceedings*. Ed. by Yannis Manolopoulos, Goce Trajcevski and Margita Kon-Popovska. Vol. 8716. Lecture Notes in Computer Science. Springer, 2014, pp. 153–162. ISBN: 978-3-319-10932-9. DOI: 10.1007/978-3-319-10933-6_12.

[36] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski and Krzysztof Stencel. "Supporting Code Review by Automatic Detection of Potentially Buggy Changes". In: *Beyond Databases, Architectures and Structures - 11th International Conference, BDAS 2015, Ustroń, Poland, May 26-29, 2015, Proceedings*. Ed. by Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek and Daniel Kostrzewa. Vol. 521. Communications in Computer and Information Science. Springer, 2015, pp. 473–482. ISBN: 978-3-319-18421-0. DOI: 10.1007/978-3-319-18422-7_42.

2 PRELIMINARIES

Contents

2.1 Software engineering 5
2.2 Information retrieval 7
2.3 Learning 8
2.4 Mining software repositories 11

In this chapter, we introduce common software engineering processes serving as a data-source for MSR purposes. Then, we describe Information retrieval and Machine learning methods used to analyze the data present in software repositories. In particular, we present related supervised learning algorithms, as well as learning to rank approaches. Consequently, we present evaluation metrics used to assess the model performance. Additionally, chapter specific notation is introduced in Section 4.3.1 and Section 5.3.1

2.1 SOFTWARE ENGINEERING

The Software engineering is a systematic approach to construct and support software. Essential software attributes are [138]:

- maintainability - the software should be written in a way that enables future changes due to changing requirements,
- dependability - the software should not cause damage in case of failure,
- efficiency - the software should not waste computing resources,
- acceptability - the software needs to be usable by users and compatible with other systems in the shared environment.

Creation of software consists of basic process activities such as specification, development, validation, and maintenance. Those activities are organized differently depending on used development methodology. If the methodology is plan driven, such as waterfall, the process activities are sequential, with each activity working on artifacts created previously, when the preceding activity is finished. In case of agile development, the activities are interleaved, and usually coordinated to develop part of a specific feature, with the goal of quickly gathering feedback and changing direction of the project if required. Each activity is a source of various kinds of heterogeneous data for MSR purposes [62].

2.1.1 *Specification and design*

During the specification activity stakeholders, architects and developers tend to work on various kinds of requirements and design documentation. The documentation is composed of natural language text describing use cases and structured documents, for instance UML

diagrams, interface description language specifications and protocol descriptions. Intuitively, such data is stored in issue tracking and project management systems. Note that the same systems are often used for bug tracking purposes, with bug reports uploaded by users[170].

2.1.2 *Development and quality*

To ensure the dependability of the developed software programmers utilize build automation tools, which handle the compilation and dependency management. Automatic test execution is usually included in configuration of such a tool, verifying if project code passes the unit and integration tests. Additional utilities such as static code analyzers, linters and coverage checkers might be also incorporated. The results are presented to the developers in the form of automatically generated text reports. The other kinds of data generated during the development activity are abstract semantic graphs and abstract syntax trees, representing the used syntax constructs and underlying structure of the project in question. The presence of ASTs enables computation of software metrics, such as cyclomatic complexity and code coverage by tests.

Version control

Version control is a system that preserves changes to a set of files with a set of metadata, enabling users to recall specific versions. The common definitions in modern distributed version control, such as git, are:

repository storage of changes of all project source files along with metadata

cloning creation of local repository based on other repository, containing the same changes

commit record of changes to the files, with unique identifier, on specific branch, with connection to parent commit and metadata such as author, time of creation and message

branch list of successive commits, enabling parallel different changes to the same files

master main branch in the repository

merge act of combining two or more branches applying all changes to the files

conflict occurs during merge when two or more changes are modifying the same part of specific file, might be resolved automatically or by explicit user input

push sending the changes from selected branch of local repository to chosen remote repository

fetch downloading changes from remote repository to local repository

pull downloading changes from remote repository to local repository and merging those changes

tag additional annotation on specific commit

Code review

The peer code reviews have been practiced since the 1970s [16, 32], recent studies show that modern code review is more lightweight and less formal than in the past [123, 127], while offering similar software quality benefits [96]. The previous technique, code inspections [32] fell to disuse due to required time constraints and formalized processes including synchronized inspection meetings. Consequently, the rise of the internet enabled developers to conduct reviews by email, based on patches sent to mailing lists [127]. Manual, error prone workload and lack of process caused development of code review systems [127].

The modern standard for both industry and open source projects is tool based review. During the code review the reviewers provide comments to the changed files, and decide if changes are ready to be merged with the codebase or need additional work. This approach utilizes the lightweight process with defined, adjustable rules, enforced by a code review system. Such rules can include a number of required approvals, specific ownership of changes and code passing build automation. Note that the pull-request model is a variant of tool based review. Benefits of code review include educating new contributors to the project, upholding standards such as chosen formatting style, high test coverage and discovery of bugs [123]. The comments and results of the review are stored in the code review system database, with corresponding changes metadata saved in the project repository.

2.1.3 *Release management, deployment and maintenance*

The release process consists of building a selected version of the project, testing and deployment of this version to the target production system in a repeatable manner. This process utilizes the same built automation tools as the development phase, but prepared artifacts, such as compiled packages, are made available to the end users [138]. The public announcements of new version, such as published change logs or email messages are the text data describing this phase. After deployment users interacting with the new version encounter the software bugs and report those to the maintainers.

2.2 information retrieval

Information retrieval is the task of finding documents that are relevant to a selected user's query [126]. For each query the IR system needs to prepare a result set, composed of documents relevant to the query from the corpus of all available documents. Document ranking, defined as sorting documents in order of likely relevance, is a main problem in IR [88].

Bag-of-words model In this model, a text is represented as orderless representation - the multiset [70] of its words. It omits the grammar structure of a document or sentence, but retains words multiplicity [126]. This model is commonly used to prepare set of features for other algorithms.

Vector space model A vector of terms represents a text in this model. As each term becomes an independent dimension in high dimensional vector space. Typically terms are words, phrases or index identifiers [135]. When a term is present in a text, it gets a non-zero value in the vector, depending on weight scheme.

TF-IDF The term frequency - inverse document frequency is a commonly used weight scheme model, representing documents as vectors of weighted terms [126], given an equation:

$$\text{tf-idf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D).$$

The term frequency $\text{tf}(t, d)$ is number of occurrence of term t in document d. The inverse document frequency represents how common is given term t in whole corpus D, following equation

$$\text{idf}(t, D) = \log \frac{|D|}{|\{d : t \in d\}|}, \tag{1}$$

where $|D|$ is the number of documents in the corpus and $|\{d : t \in d\}|$ is the number of documents containing term t.

Topic model The topic model represents a document as a series of probabilistically generated topics [53]. A Latent Dirichlet Allocation [17] is a three-level hierarchical Bayesian topic model. Each document is modeled as a finite mixture over an underlying set of topics, and each topic is modeled as an infinite mixture over an underlying set of topic probabilities [17]. The probabilities of topics provide a document representation. This model works under assumption that topic distribution has sparse Dirichlet prior, corresponding to intuition that documents encompass only a small set of topics and that topics use only a small set of words.

Similarity measure The similarity measure is a function defined on two objects in specific domain, with codomain of real numbers. Many similarity measures have been proposed, including inverse of various distance metrics, information content, mutual information, Dice coefficient, cosine coefficient and feature contrast model [86]. Most of those measures increase with commonality and decrease with difference of objects in question, under the assumptions that the more commonality objects share the more similar they are. The maximum similarity between two objects is 1, and is reached when objects are identical.

2.3 LEARNING

An agent is an entity that can be viewed as perceiving its environment and acting upon that environment. The behavior of the agent is defined as agent function. Consequently, a learning agent improves performance on future actions after observing the results. We may distinguish between several kinds of learning, based on feedback usage or lack of thereof. Those kinds are unsupervised learning, reinforcement learning, supervised learning and semi-supervised learning [126]. In this dissertation we focus on supervised learning and learning to rank applications as described in the following sections.

2.3.1 *Supervised learning*

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs [126]. Given a training set of $|N|$ examples, $(x_1, y_1), (x_2, y_2)..., (x_N, y_N)$ where x_i denotes input, and y_j was computed by an unknown function f such that $y_j = f(x_j)$, the task of supervised learning is to find a hypothesis function h, which approximates f [126]. To verify h a separate testing set is used. If

y_j belongs to a finite set of discrete values the learning task is classification, identifying to which of a set of categories a new observation x_j belongs. On the other hand the regression is a problem of estimating continuous variable y_j by independent variables of observation x_j.

Linear regression Let x_j be m element vector. The hypothesis function takes form

$$h_w(x_j) = w_0 + \sum_{i=1}^{m} w_i * x_{i,j},$$

where w is a weight vector. To find optimal hypothesis function the w vector needs to be changed according to the selected learning approach, based on results on the training set. Let squared loss function be

$$L_2(h_w) = \sum_{j=1}^{N} (y_j - h_w(x_j))^2.$$

Given N training examples, the sum of the individual losses for each example needs to be minimized. The best set of weights is

$$w^* = \arg\min_{w} \sum_{j=1}^{N} L_2(y_j, h_w(x_j)).$$

Thus each weight can be updated according to equation:

$$w_i \leftarrow w_i + \alpha \sum_{j=1}^{N} x_{j,i} * (y_j - h_w(x_j)),$$

where α is the learning rate, until minimum loss is found.

Logistic regression To utilize regression for classification problems logistic regression can be binomial, multinomial or ordinal, depending on possible y_j values. Intuitively, for binomial logistic regression $y_j \in 0, 1$, multinomial logistic regression deals with situations where y_j can have three or more possible types that are not ordered. In case of ordinal regression dependent variables are ordered. Let

$$l(z) = \frac{1}{1 + e^{-z}}$$

be the logistic function. The hypothesis function is then defined as

$$h_w(x_j) = l(w * x_j) = \frac{1}{1 + e^{-w * x_j}}.$$

Support-vector machine A Support-vector machine model is a representation of the training set as points in space, and dividing them by maximum margin separator hyperplane, which is a decision boundary with the largest possible distance to training points [126]. Test set is mapped into the same space, with predictions based on the side of the hyperplane of each point.

For $y_j \in 1, -1$ the hyperplane is defined as

$$w * x_j - b = y_j,$$

where weights vector w is normal vector of hyperplane, and b is the intercept parameter.

The hypothesis function is defined as

$$h_{w,b}(x_j) = \begin{cases} 1 & \text{if } w * x_j + b > 0 \\ -1 & \text{otherwise} \end{cases}.$$

To find the separator hyperplane the following equation is solved [126]:

$$\arg\min_{w,b}(1/2 * w * w + \sum_{j=1}^{N}(max\{0, 1 - y_j * (w * x_j + b)\})).$$

This model was adapted for text categorization due to the fact that it is both able to work efficiently with high dimensional input space and sparse vectors representing documents [59].

2.3.2 *Learning to rank*

Learning to rank is the process of applying machine learning to the ranking problem in information retrieval systems. The approaches to learning to rank can be divided according to hypothesis function domain and codomain into three main approaches [89].

In the pointwise approach the hypothesis function is defined on feature vectors representing single documents. The corresponding output is the relevance degree of a given single document [89]. Thus the hypothesis function can be modeled as regression, classification, or ordinal regression, with corresponding loss function checking the prediction for each single document.n of the ground truth label for each single document.

The domain of hypothesis function in the pairwise approach are the pairs of feature vectors representing documents. The result is the pairwise preference $+1, -1$ of order between documents. Such ranking is modeled as a classification task, with loss defined as classification loss [89].

For the listwise approach the hypothesis function is defined on sets of documents related to the query, with output being permutation of documents [89]. The corresponding loss function is defined with respect to all the documents associated for a given query.

2.3.3 *Model performance metrics*

The following are common metrics used in most studies to assess performance [54, 57, 79].

Let *actual*(n) be the actual relevant n documents for specific query and *top*(n) be the top n documents retrieved for the same query. *Precision@k* represents an estimation of how many documents are correctly recommended for specific query within given top k. It is also known as positive predictive value. It is computed as follows:

$$Precision@k = \frac{|top(n) \cap actual(n)|}{|top(n)|}. \quad (2)$$

Recall@k estimates how many documents are correctly recommended within given top k over the actually relevant documents. It is also known as true positive value. It is computed as follows:

$$Recall@k = \frac{|top(n) \cap actual(n)|}{|actual(n)|}. \quad (3)$$

F1 score is the weighted harmonic mean of precision and recall for given k. It is also known as *F-Measure*.

$$F1 = 2 * \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{4}$$

Accuracy@k, also known as likelihood, measures the percentage of queries for which the model predicted at least one correct recommendation in the top k ranked documents and is defined as:

$$\text{Top K} = \text{\# at least one correct in top } k, \qquad Accuracy@k = \frac{\text{Top K}}{|Q|}. \tag{5}$$

Mean Average Precision for a set of queries is the mean of the average precision scores for each query, given an equation:

$$MAP = \sum_{q \in Q} \frac{AvgP(q)}{|Q|} \tag{6}$$

where the average precision *AvgP* score defined as

$$AvgP = \sum_{k \in K} \frac{Precision@k}{|K|}.$$

Mean Reciprocal Rank is the metric commonly used in the information retrieval ranking problems [152]. It aids evaluating any process that produces a list of possible responses to a series of queries, ordered by the probability of their correctness. This measure computes the mean value of the position of the first relevant document in the recommended ranked list:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\text{rank}\big(first(q)\big)}, \tag{7}$$

where $\text{rank}(first(q))$ is the position of the first relevant document in the ranked list for query q.

2.4 MINING SOFTWARE REPOSITORIES

In this section we provide examples of MSR corresponding to selected phases of software engineering. One of the main MSR goals is enabling architects, programmers and coordinators to make informed decisions about developed projects, based on the available historical data, without a need to depend only on their intuition [47]. Depending on the required usage, the information retrieval and machine learning algorithms need to be applied on the cross linked data, with results presented as feedback to potential users [62]. On the other hand, various repetitive or time consuming development activities can be automated, enabling contributors to focus on other work.

2.4.1 *Augmenting feature specification*

Defect detection might be hampered by misclassification of features as bugs [71]. Tools like ReLink[160] and Linkster[8] help to create links between bug reports and commits, either automatically or manually.

Ray et al. [121] investigated software quality of 729 open source projects hosted on GitHub. The authors checked the impact of language on quality using features such as procedural/functional/scripting paradigm, strong/weak type system, static/dynamic typing, managed/unmanaged memory model and scripting/compiled model. Moreover, they used regression on specific project features, like size, project history, number of contributors to estimate the number of defects per project. In addition to that both projects and defects were also clustered to find relations between defect types and languages. The authors concluded that functional languages have smaller relation to defects than other paradigms, some languages are more associated with defects and kinds of present defects are strongly associated with the used language.

Marcus [93] attempts to find similarities between user queries and source code by using concepts obtained via Latent Semantic Indexing. The method is trained on corpus prepared from source code of Mosaic web browser and set of artificially created queries, and evaluated against "grep" like tool, with better results in terms of recall.

Bajracharya [9] proposes the method to find examples of api usage by utilizing similarity between api implementations using Structural Semantic Indexing.

2.4.2 *Automated development*

Kim et al. [65] proposed a patch generation tool named "Pattern-based Automatic program Repair". This approach is based on application of fix templates to suspicious code, found by fault localization algorithms. Each generated program variant is evaluated using fitness function computing the number of passing unit tests. Authors manually prepared generic fix templates based on 62656 human written patches of Eclipse JDT project. During evaluation the tool was able to prepare fixes for 27 out of 119 from six open source projects: Mozilla Rhino, AspectJ, Apache Log4j, Apache Commons subprojects Math, Lang and Collections.

Jia et al[58] introduce higher order mutants by applying mutation operators more than once as an extension to mutation testing. Authors tested four approaches (fitness function, greedy algorithm, genetic algorithm and hill climbing algorithm) on six test programs written in C. Best results were obtained using hill climbing algorithm.

Balog et al [11] use neural network to compose programs solving programming competition problems. The network composes programs from available functions and higher order functions (map, filter, reduce) to pass the competition criteria.

2.4.3 *Code review enhancement*

Modern tool based code review already utilizes feedback generated by compilation and unit test results. Analysis of code review results, addition of new sources of feedback and further automation of reviewer activities is an area of study within MSR. We analyse in detail state-of-the-art reviewer recommendation approaches [10, 145, 167, 168] in Section 4.2.

Rigby et al. [122] analyzed convergent contemporary peer review practices across Microsoft, AMD, Google and Lucent. Majority of reviews involve two reviewers. The modern code review tools improve traceability and information sharing over traditional review processes and are widely adopted in industry.

Paixao et al [114] introduced the Code Review Open Platform, the curated repository linking review data with source code snapshots, mined from 8 open source projects using the Gerrit code review system.

2.4.4 *Bug reports handling*

Duplicate bug reports cause developers to waste time. Li et al. [85] proposed to detect duplicate pull requests on GitHub via information retrieval model. The method is based on computation of cosine similarity on textual descriptions of each pull request, represented by Vector Space Model.

Different approach was proposed by Lazar et al. [76]. The authors evaluated algorithms available in scikit learn (K-Nearest Neighbours, Linear SVM, RBF SVM, Decision Tree, Random Forest and Naïve Bayes) and LibSVM with a goal of binary classification (duplicate/non duplicate) on bug reports of 3 open source projects Eclipse, Open Office, and Mozilla. Features were obtained by TakeLab. Best results were returned by LibSVM and K-Nearest Neighbours.

2.4.5 *Defect fighting*

Bhattacharya et al. [13] conducted an empirical analysis of the bug fixing process in 24 open source Android applications and Android platform itself. The authors used Bugzilla instances of public Google Code repositories, choosing applications with more than 200 bug reports and more than 100000 downloads. The gathered data was used to measure the bug report quality, distribution of report statuses, developer teamwork and required time to prepare a fix per each project. The negative correlation was discovered between features describing report description length and time to fix within months, which led the authors to the conclusion that length is a good predictor of report quality. The average bug fixing time is less than 1.5 months, and developer communities start work within one day from new report creation, mostly by adding comments.

We can distinguish two main approaches to detection of bugs, one based on a specific single version of source code such as selected release and the other utilizing repository data for project history[25].

Moreover, such detection can be performed on different levels of granularity. *Spectrum-based fault localization*[139] and *Probabilistic fault localisation* [75] both assign a degree of suspicion to each source code line. *Savant* [77] tool selects methods most likely to contain defects. Those methods with sub-file granularity level are based on execution traces, often gathered from test suites from selected single version of code. *Network Analysis on Dependency Graphs* by Zimmermann et al. [173], *Static Code Attributes* by Menzies et al. [99] and the *Ant colony optimization model* by Vandecruys et al.[149] all detect defects on component or module level, utilizing different methods. Arisholm et al[5] assessed how different modeling techniques can affect fault prediction algorithms. They investigated feature engendering based on various object oriented and process metrics, model selection and evaluation criteria compared against each other using cost-effectiveness. The features based on object oriented metrics do not yield good results when compared with others. The differences introduced by various modeling techniques are insignificant.

The replicability of research results is one of the main concerns in this field, as only 31% of fault detection papers published before 2009 used public datasets according to Catal et al. [19] survey.

2.4.6 *Bug localization*

Various models were proposed to localize a known bug, depending on the user supplied report [2, 22, 104, 109, 128, 131, 158, 165, 172]. Some of the previously mentioned bug localization methods use various learning-to-rank algorithms; thus, we present the application details. The Random Forests and MART algorithms from RankLib [27] evaluated by Shi et al. [131] are examples of *pointwise* learning-to-rank applications. Ye et al. [164, 165] use of SVMrank is the example of *pairwise approach*. Shi et al. [131] tested some *pairwise algorithms*: RankNet, RankBoost and LambdaMART. *Listwise* ranking algorithms were utilized in AmaLgam+ [155] and CoordinateAscent used by Shi et al. [131]. In Section 5.2 we provide a detailed description of those state-of-the-art approaches.

Lee et al [78] conducted replicability study on BugLocator, BRTracer, BLUiR, AmaLgam, BLIA and Locus bug localization algorithms [128, 156, 158, 166, 172], preparing new dataset containing 9459 bug reports 46 projects of Apache, Spring and Jboss. Surprisingly, duplicate bug reports present in the dataset help with obtaining better accuracy by inclusion of additional tokens complementing the main bug report and none of the six evaluated methods is able to outperform the others.

Bird et al. [14] analyzed bug fix datasets in terms of bug feature bias and commit feature bias. The authors used source code of AspectJ and Eclipse together with iBugs dataset linking changes to bugs to train BugLocator [172]. Evaluation results show that a large number of less severe bugs present in datasets cause localization algorithms to better find such bugs than critical or blocking ones. When training only on a specific kind of bug, the algorithm works well for the selected kind, but has worse results for others.

3 PRELIMINARY STUDIES ON OPEN SOURCE DATA

Contents

3.1 Introduction 15
3.2 Related work 15
3.3 Contribution analysis 18
3.4 Bug detection based on commit similarity 20
3.5 Concluding Remarks 22

3.1 INTRODUCTION

In this chapter, we describe the results of preliminary studies conducted on GitHub open source projects. In particular, we investigated developer roles, their participation scope, maintenance work management and method for bug detection for code under review. During the normal development process some contributors leave the project, and new arrive, resulting in a shift of domain knowledge. Thus, we analyzed commits of open source projects using topic modeling to capture roles of developers and scope of their work. To this end, we utilized a subset of the GHTorrent dataset [41]. Consequently, we identified bug fixing as one of the most prevalent activities. Therefore, inspired by Kim et al. [67] we prepared a code review bug classifier to help project maintainers.

This chapter is based on ideas from our two published conference papers [35, 36]. The contributions are as follows:

- we conduct an open source contribution analysis on 42 projects and define developer groups responsible for most of the work;
- we prepare a classifier detecting bugs during code review in git repositories, and evaluate it on 64 projects.

3.2 RELATED WORK

In this section we introduce publications related to open source contributions and detection of software bugs.

3.2.1 *Developer collaboration*

Mockus et al. [101] analyzed development and maintenance of major OSS projects: the Apache server and Mozilla by mining email archives. They investigated the processes used to develop projects, percentage of people per role within the project, if work is evenly split and the defect density. The Apache server development is guided by a set of "core developers" volunteers also known as Apache Group who contributed to the project for

an extended period of time. Those "core developers" identify work to be done, such as missing features or defects, and orchestrate it. There is no defined development process, but new changes are reviewed by Apache Group who have implicit ownership of parts of the codebase. Developers communicate mostly by mailing lists. Usually one of the core team prepares a release. In the time of the study core team had 25 developers compared with 400 non core team contributors. The Mozilla also has 12 core staff members, most of the work is delegated to 500 volunteers. Core staff are designating module owners from volunteers, who have a blocking right during code reviews. The releases are also coordinated by core staff. Both new features and defects are reported on Bugzilla. Based on both projects authors formulate hypotheses that open source projects have a core team responsible for 80% of the new functionality.

Scialdone et al [130] investigated Free/libre Open Source Software teamwork behaviors, distinguishing between core and peripheral members. They studied the development of two instant messaging clients: Gaim and Fire using 45 months of public email archives. Each message was converted to *Group Maintenance Indicators* such as "Encouraging participation", "Formal verbiage", "Expressing agreement" divided into 3 categories: Emotional Expressions, Positive Politeness and Negative Politeness. Indicators per each group (core/peripheral developers) were compared using Mann-Whitney U test. Positive politeness behaviors were more common in Gaim, while negative politeness was more common in the Fire community. The core developers of both projects more commonly expressed a sense of belonging within their groups.

Breu [18] analyzed bug reports in terms of reporters and programmers collaboration. The authors suggest creating interactive tools like community driven portals engaging both developers and users. The goal is to help with unanswered questions on both sides like missing replication data or status updates on fix.

Murphy-Hill et al.[105] analyzed Microsoft developers work during fixing activity by interviews (40 participants) and questionnaires (362 participants). Answers show that developers tend to create fixes with as few lines as possible, with maintaining the original design and backwards compatibility.

3.2.2 *Bug detection*

The history of bug fixes can be used to train defect detection models that try to predict which files are buggy based on their contents and history. Zhang et al. [169] aimed to create a universal defect detection model. Micro Integration Metrics [80] and Change Bursts[106] try to capture developer interaction with source code files. Palomba et al. [115] added code smell features with JCodeOdor tool. In the work of Moser et al. [104] the goal was to find which type of metrics are best suited to detect defects. Jaafar et al. [56] analyzed the relationship between defects and anti patterns. Ostrand et al. [111] aimed to predict which source code files have the highest number of defects.

Localization via code similarity

Pan et al.[116] analyzed patterns in software defects in 7 large Java projects (ArgoUML, Columba, Eclipse, JEdit, Lucene, MegaMek, Scarab), categorizing them into 27 kinds of bugs. The author notices that most frequent patterns (like "method call with different parameter values" or "change in if conditional") have similar frequency across projects.

Livshits et al [91] proposed the *DynaMine* tool to discover method usage patterns and violations of those patterns. For example chained methods used to setup network connection constitute such a pattern. The source code is mined using Apriori algorithm to find association rules between source entities. The tool was evaluated on two large Java applications Eclipse and jEdit and found 250 pattern violations.

Couto et al[24] prepared visualization tool based on Granger Test results to localize defects. The method is based on a time series of twelve source code metrics calculated on class level. The tool was tested on Equinox Framework and Eclipse JDT Core open source projects.

A related problem is considered by the authors of *DebugAdvisor*[7], who proposed a search tool able to find already solved bugs similar to the currently modified code. This tool is able to search several sources like bug databases, software repositories and program traces via converting content to "typed documents" consisting of bags of terms, ordered lists of terms, weighted terms and key-value pairs. Retrieval of appropriate documents is based on weighted TF-IDF. The tool was evaluated by questionnaire on 100 Microsoft employees.

Kim et al. [69] proposed to identify bug-introducing repository changes using annotation graphs consisting of multiple commits instead utilizing only annotations of the removed lines as in the Śliwerski algorithm [136]. The algorithm was evaluated on Columba email client and Eclipse jdt.core against the previous state-of-the-art removing 48% of false positives.

In the following paper Kim et al. [67] introduces change classification as a method to find latent bugs. The authors trained the SVM classifier on defect commit data, using seven groups of features: added delta, deleted delta, changed file paths, change log, new files, commit metadata and complexity metrics, obtaining multiple changes from single revision. The method was evaluated using *accuracy*, *precision*, *recall*, and *F1 score* against a dummy classifier on 12 open source projects, using 500 and 250 last revisions for training, with better results.

Lewis et al [84] interviewed 19 Google developers if they found bug localization tools FixCache and Rahman useful. Both tools are used within Google to mark bug-prone files during code review. Although tools are correctly detecting such files according to the developers feedback, some programmers complain that there is no suggestion how to fix such files.

Shivaji [133] utilizes feature selection techniques on source code datasets [66] consisting of code metrics, code metadata and source code converted to bags-of-words to find the best features to train the SVM classifier and Naïve Bayes. From the history of 11 projects as much as 3125 defect changes and 9294 non-defect changes are extracted with total 183054 features. Methods using feature selection outperformed Kim [66] in terms of *precision*, *recall* and *F1 score*.

Shin et al. [132] conducted an empirical study to predict faults using calling structure information. Authors used 30 releases of industrial projects to evaluate several negative binomial regression models. Used features were based on file level using history, code attributes and calling structure (callers and callees of each method). Addition of calling structure provided marginal improvement. Models using only history features obtained better results than others.

Ostrand et al. [112] conducted a study on 16 releases of industrial software system in order to find if inclusion of authorship information features can improve defect predictions

of metrics based model. The defect prediction is based on negative binomial regression trained on features such as source file age, logarithm of number of lines. The source code of this 35 years old industrial system is not available, nor is the source code of the method. The authors conclude that inclusion of authorship information improves sightly prediction results. The method was able to select 20% of files responsible for 75% of faults during evaluation.

Linares-Vasquez et al. [150] propose to select maintainers to triage incoming changes based on code authorship information. For a given change request, the authors use Latent Semantic Indexing to find related classes from a single version of the analyzed project. Then the maintainers are selected utilizing authorship information present in class source comments, with authors ranked by occurrence frequency among found files. The method was evaluated on ArgoUML, jEdit, and MuCommander open source projects using precision and recall metrics against Anvik et al. [4] approach and xFinder method, and was able to outperform other methods on jEdit and MuCommander projects.

Similar problem was investigated by Anvik et al. [4], with the goal of bug report assignment to a developer, based on specific knowledge and history of already solved reports per developer. The method is based on training multiple classifiers, each for selected developer. The authors tested SVM, decision tree and Naïve Bayes using normalized feature vectors indicating the frequency of the terms in the bug report text, choosing SVM using precision and recall metrics. The training was done using two open source projects: Eclipse IDE and Firefox browser, with evaluation on gcc.

Localization utilizing test execution

Jones et al. [61] compares the *Tarantula* fault-localization tool with localization via set union, set intersection, nearest neighbor and cause transitions techniques on Siemens test suite, with better results for Tarantula.

Wong et al. [159] proposed to use a radial basis neural network to localize defects. The network is trained on test coverage information and test result (success or failure) and returns suspiciousness score per each code statement. Authors used four C programs for evaluation: Unix Suite, Space, Make, Grep and one Java program Ant. Faulty versions of programs were created by usage of mutation testing.

To find bugs Liu [87] proposed a probability based method which analyses boolean predicates in run traces. Faulty executions are observed to have different distributions of predicate values than non faulty ones. The tool was evaluated on a Siemens suite of 130 bugs against methods CT and Liblit05 with better results.

3.3 CONTRIBUTION ANALYSIS

In our experiments we focused on investigation of commits contributed to open source projects. We used the GHTorrent dataset [41], which contains data of 90 GitHub projects and their forks, out of which we choose 42 projects containing enough issues and commit comments for our purposes. The dataset details are shown in Appendix Section 7.2 in Table 7.1 and Table 7.2. To discover areas of interest we selected the Latent Dirichlet Allocation implementation from *Mallet* topic modeling toolkit [95]. Consequently, we extracted topics separately for issues and commit comments per each project. Each issue or comment

Table 3.1: Statistics of commit commenter groups

Definition	Commenters					Commiters
	Specialists	Generalists	Both	Other	Total	
Total	77	412	73	1065	1481	14775
Average per project	1.83	9.81	1.74	25.36	35.26	351.79

Table 3.2: Core contributors

Project	Team on web page	Both	Generalists
django	26	8	15
jekyll	7	4	4
jquery	107	9	14
libgit2	74	3	7
scala	27	4	10

was treated as a single document. Due to the prevalence of sentences like ”thank you” we adjusted stop words to rule those out.

3.3.1 *Core team detection*

In order to understand contributor roles within open source projects we investigated the commit commentary topics aggregated per commenter. Our goal was to gather insight on code review activity. We define two main kinds of commenters - specialist and generalist. A specialist is a contributor, whose number of comments matching a specific topic is larger than half of the maximum number of comments to the most popular topic of a project. A generalist is a contributor, whose number of comments matching multiple topics is larger than the average number of topics per committer. Note that it is possible for one contributor to belong to both groups simultaneously.

Work is not distributed evenly, and only 13.21% of contributors provide comments on the work of others, as shown in Table 3.1. The smallest group of specialists only comment on the most significant topics. Almost all of the specialists are also generalists, providing comments to a number of topics. We investigated web pages of 5 selected projects to check if generalists are mentioned on the project team web page. As shown in Table 3.2 our approach is able to find a raw estimate of which contributors are team members responsible with reviews, based on their comments.

3.3.2 *Defect prevalence*

Investigation of contributors leads us to question if the bug fixing activity is common in analyzed projects. We considered a topic related to this activity wherever it contained words "bug", "fix" or "solve". In order to assess distribution of bug related topics we utilized aggregation per month for both issues and commit comments.

The analyzed projects had issues matching the bug fixing topic present in 96% of analyzed months. For those months 91% of issues matched at least one of bug related topics.

In case of commit comments, the bug topic was present on average during 64% of analyzed months across all projects, with 54% comments matching at least one of bug topics.

3.4 BUG DETECTION BASED ON COMMIT SIMILARITY

Based on previous results, showing that most contributors do not provide comments to commits and that defects are common, we decided to focus on preventing introduction of buggy changes on code review level. To integrate code changes with the destination branch on the remote repository developer typically creates a separate branch for code review purposes. This branch is handled by a code review system, which displays the difference between changed files and destination branch to reviewers, and allows reviewers to add comments, based on their knowledge and past experience with the project. Reviewers notice suspicious changes, introducing bugs or bad practices, but some might be omitted due to time constraints or lack of knowledge about project history. Within legacy projects the original authors tend to be not available as reviewers.

3.4.1 *Proposed solution*

For each opened review in the code review system our method assesses each modified file using a classifier trained on past defects. Following Kim et al. [67] proposition to classify changes we use SVM classifier [59], and provide results to reviewers as comments. The main differences between our approach and Kim et al. are reduced number of features, marking the whole commit as buggy/non buggy without splitting into separate files and weighting scheme balancing disproportion between classes in the training set. We train the algorithm only on data already present in the repository, without querying outside sources.

Model training

Kim et al. [67] change classifier utilizes features based on change metadata, change log messages, source complexity metrics, changed file paths and code delta (difference). We decided to use only a subset of features corresponding to data directly available to the reviewer, the source code delta, focusing on the problematic code, not on its metadata such as an author or time of creation. Additionally, we do not treat the commit as multiple changes per each file. To prepare a dataset for training we parse results of git commands using parser combinators [102]. Consequently, we convert the commit changed delta to "bag of words" model.

Note that, to train algorithm we require information about fixed bugs. For this purpose we use the same approach as Kim et al. The *Śliwierski-Zimmerman-Zeller algorithm (SZZ)* [136] is an MSR information retrieval technique for annotating bug commits via finding fixing commits. It searches commit messages for keywords such as "bug", "fixes" or external bug report system id. Such commits are considered fixes, and commits changing the same lines as fixes are considered buggy. Intuitively, we use the *SZZ* approach to mark a selected number of commits for model training purposes. In particular, we denote this number as the *history limit*. We do not adjust the keywords manually per project.

In order to train the SVM classifier we utilize the Sequential Minimal Optimization implementation provided by Weka toolkit [44]. Due to an imbalance between buggy and non

buggy data in the training dataset, depending on *SZZ* algorithm results, we adjust the weights of instances, so both classes have the same total weight.

Model usage

The resulting classifier is used in the future when new commits are under review. Each change the classifier is applied to the corresponding "bag of words" of commit delta. Results are sent to the review system as comments. We provided integration with the Gerrit code review system.

3.4.2 *Evaluation Results*

In this section we present the empirical evaluation of the utilized model. The Kim et al. [67] implementation working with cvs repositories and corresponding dataset are not available. Consequently, we prepared our own dataset containing selected open source git repositories. We calculated the *accuracy*, *precision*, *recall* and *F1 score* metrics per each class (buggy, non-buggy) in order to examine our model, according to the definitions as in Section 2.3.3.

We established our selection of projects based on the number of commits, length of development history and used language, with the goal to have a diverse dataset of nontrivial, actively developed applications. We divided our experimental evaluation into following three groups of tests:

1. preliminary tests, conducted on 10 GitHub repositories, using last 500 commits,
2. history length tests, run on 63 GitHub repositories and converted Subversion repository, using different history limits,
3. additional history limit tests, conducted on 10 large GitHub repositories.

Preliminary investigation

Our goal was to investigate the method utilizing reduced number of features compared to Kim et al. approach [67]. In order to test the versatility of the proposed approach we selected ten projects based on available history length. We investigated the recent changes to the repositories, and retrieved changes for evaluation from the last 500 commits. Evaluation changes are split into 400 commit training set and 100 commit testing set. Results presented in Table 3.3 show that the method is able to achieve good results without manual tuning per project.

Project history analysis

We applied the method to the selection of open source GitHub projects, using history limits of 500, 1000 and 1500 last commits respectively, using training size of 80% in each case. In this experiment we focused on the relation between length of project history and method ability to detect bugs, as more mature projects tend to have more maintenance or bug-fixing changes, affecting the training set construction. To this end we divided projects into 4 size groups using a number of commits, as presented in Table 3.4. Additional details of project groups are available in Tables 7.3, 7.4, 7.5 and 7.6 of Appendix Chapter 7.

The metrics results per each group shown in Table 3.5 show that higher history limit results in better detection of bugs, corresponding to *F1 score* for buggy class.

Table 3.3: The statistics of preliminary tests as trained on recent changes

Project	Accuracy	Buggy			Non Buggy		
		Precision	Recall	F1	Precision	Recall	F1
flockdb	0.85	0.941	0.533	0.681	0.831	0.986	0.902
gizzard	0.89	0.733	0.611	0.667	0.918	0.951	0.934
hiphop-php	0.83	0.818	0.375	0.514	0.831	0.974	0.897
jquery	0.66	0.727	0.678	0.702	0.578	0.634	0.605
sbt	0.83	0.733	0.458	0.564	0.847	0.947	0.894
libgit2	0.83	0.571	0.222	0.32	0.849	0.963	0.903
akka	0.77	0.588	0.385	0.465	0.807	0.905	0.854
django	0.77	0.795	0.674	0.729	0.754	0.852	0.8
cakephp	0.76	0.455	0.217	0.294	0.798	0.922	0.855
mono	0.70	0.357	0.192	0.25	0.756	0.878	0.813

Table 3.4: Groups of GitHub projects by history length

Group name	Number of Commits		
	min	max	avg
Small	740	2005	1344
Medium	3084	6421	4450
Large	7198	21529	11273
XL	24045	120396	49871

To further investigate the impact of history limit on training results, we conducted detailed evaluation on 10 projects, two from medium history length group, four from large and four from xl groups respectively. The history of all projects in question is longer than 5000 commits. We selected additional history limits of 2500 and 5000, using the same proportions between training data and testing data as before. For largest projects utilizing higher history limits such as 5000 or 2500 returns the better results in terms of *F1 score*, as shown on Figure 3.1.

3.5 CONCLUDING REMARKS

Based on contributor comment analysis we discovered that most contributors are not involved in comments exchange related to commits. Furthermore, our method is able to select the main contributors per project. According to topic aggregation per date the majority of reported GitHub issues are bugs, and more than half commit comments are related to bug fixing.

Our evaluation of bug detection method shows that the approach is generally applicable to projects of various history length and different programming languages. The algorithm is able to achieve good results using change delta features, and the same set of bug detection patterns for *SZZ* algorithm across all investigated projects. The results are not directly

Table 3.5: Average metrics observed in each repository size group

Group name	History limit	Accuracy	Buggy			Non Buggy		
			Precision	Recall	F1	Precision	Recall	F1
Small	500	0.728	0.552	0.355	0.425	0.751	0.871	0.806
	1000	0.710	0.679	0.456	0.541	0.713	0.846	0.773
	1500	0.708	0.672	0.477	0.553	0.704	0.829	0.76
Medium	500	0.772	0.535	0.319	0.394	0.79	0.902	0.842
	1000	0.742	0.616	0.411	0.484	0.76	0.863	0.807
	1500	0.724	0.612	0.415	0.487	0.723	0.844	0.778
Large	500	0.818	0.55	0.276	0.361	0.843	0.939	0.888
	1000	0.789	0.507	0.271	0.351	0.819	0.928	0.869
	1500	0.765	0.534	0.313	0.393	0.796	0.907	0.848
XL	500	0.838	0.461	0.219	0.284	0.862	0.945	0.901
	1000	0.810	0.522	0.244	0.327	0.833	0.943	0.884
	1500	0.789	0.51	0.264	0.347	0.813	0.927	0.866
On average	500	0.788	0.528	0.296	0.371	0.811	0.914	0.859
	1000	0.762	0.58	0.345	0.426	0.779	0.894	0.831
	1500	0.744	0.582	0.367	0.444	0.758	0.876	0.812

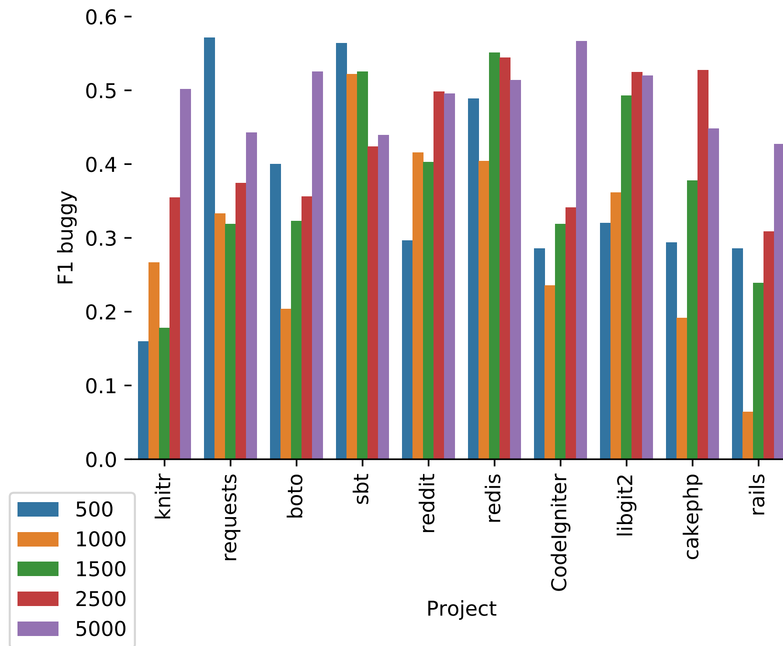


Figure 3.1: The F1 score results for buggy class with history limit set to 500, 1000, 1500, 2500 and 5000

comparable with Kim et al. [67], due to the usage of different dataset, consisting of a larger number of projects and disparate treatment of code changes in corresponding repositories.

However, while our approach obtains a lower average *F1 score* than Kim et al. [67], it is able to achieve a comparable average *accuracy* of 0.788 on all projects for the history limit 500. Further increase of the history limit according to the relative project history causes better detection of defects, based on higher *F1 score* for the buggy class.

4 RECOMMENDATION OF CODE REVIEWERS

Contents

4.1 Introduction 25
4.2 Related work 25
4.3 Problem statement 29
4.4 The proposed method 30
4.5 Evaluation Results 35
4.6 Discussion 43
4.7 Concluding Remarks 44

4.1 INTRODUCTION

In this chapter, we focus on the automation of the code reviewer recommendation. This problem has been intensively studied, with various algorithms being proposed. Thongtanunam et al. [145] introduced the Revfinder tool, measuring the similarity of commit file paths to already done reviews. Balachandran proposed to use repository authorship information on file level to select reviewers, creating Review Bot [10]. Yue Yu et al. utilizes both text similarity and traversal of developers interaction graph to assign reviewers [167, 168].

We propose a novel algorithm, working without disadvantages of previous methods, such as high computational complexity of file path comparison in Revfinder and low accuracy in Review Bot. This chapter is based on a published journal article [34]. Our contributions are as follows:

- we propose a novel method to select reviewers in the code reviewing process;
- we perform thorough experimental evaluation of this method;
- we compare this method with state-of-the-art techniques to prove its quality.

4.2 RELATED WORK

In this section we present an overview of literature directly related to the problem of assigning reviewers. As this problem was first under consideration without software development context we present several historical approaches to select reviewers for research papers.

4.2.1 *Reviewer Assignment Problem*

The selection of appropriate reviewers, known as Reviewer Assignment Problem has been intensively studied [153]. Various models, based on information retrieval, natural language

processing, graph analysis, recommender systems, supervised learning and clustering had been proposed in this research area [23, 153].

Dumais et al [30] proposed to use Latent Semantic Indexing to match documents with potential reviewer biographies.

Yarowsky [163] experimented with several methods to route conference papers to reviewers and area committees. The first of those models is the cosine similarity on normalized tf-idf word vectors. The vectors are obtained from papers published by reviewers and from the paper under review. The second is Naïve Bayes classifier, trained on the same data. The third and fourth methods are utilizing transitive bibliographical similarity, defined on paper citations features. In both cases similarity function is based on the number of common citations, either between all paper coauthors and potential reviewers for the third model, or between corresponding authors and reviewers for the fourth. All methods were evaluated against four human judges on a dataset of 92 real research papers. The author-reviewers transitive bibliographical similarity obtained the most agreement with human judges, with cosine similarity as a second best method.

Conry et al. [23] proposed latent factor collaborative filtering to suggest conference paper reviewers. In this model each paper or reviewer is characterized as vectors in space, and interaction (act of review) is defined by the inner product in that space. The authors utilize reviewer to reviewer similarity defined as number of same coauthors and paper to paper similarity obtained from the cosine similarity of abstract word vectors. The method was evaluated on 529 papers and 203 reviewers against the Taylor affinity graph model [142], with better results.

Toronto Paper Matching System [20] is a reviewer suggestion system used for machine learning and computer vision conferences. The selection of reviewers is based on similarity between reviewer profile, constructed from reviewers previously published papers and paper under review. The authors construct profiles either from normalized word vectors, or from topics obtained via Latent Dirichlet Allocation. The similarity score is then used to train a supervised prediction model. Available models are: Linear Regression (with or without shared parameters), Probabilistic Matrix Factorization and Restricted Boltzmann Machine. This system was evaluated on two datasets: NIPS 2010 containing 1251 papers and 48, ICML-12 containing 857 papers and 431 reviewers. The Linear Regression with shared parameters trained on word vectors profiles obtained lowest Root Mean Square Error on both datasets.

Xie et al. [161] prepared a mathematical model formalizing the reviewer selection process. The model is used to assess the number of required reviewers reviews per paper in four conference tiers (low, medium, prestigious and starting/unknown) based on distribution of estimated intrinsic quality of paper. The authors estimate at least three required reviews per paper for medium tier conference, and at least seven for prestigious one.

Tang et al. [141] proposed to select research paper reviewers based on an expertise graph. In this method the reviewer assignment problem is transformed to find convex cost flow. The nodes in the graph represent queries and domain experts, edge weights are calculated from topics common to both query and expert. The topics are obtained from Latent Dirichlet Allocation. The authors evaluated the graph walk method on KDD'09 dataset containing 338 papers and 354 reviewers against the Greedy algorithm as a baseline, with better results.

Liu et al. [90] introduced a traversal of expertise graph using Random Walk with Restart. The graph is prepared using information about expertise, authority and diversity. Similar

to Tabg et al. [141] the graph nodes represent queries and domain experts, with weights calculated using cosine similarity on topics prepared via Latent Dirichlet Allocation. The method is evaluated against text similarity, topic similarity and Random Walk without Restart using *precision@k* metric, and outperforms them.

Our code review recommendation method is not inspired directly by any of those approaches. However, we build our proposal from well-known elements (like reviewer and document profiles) and adapt them to new contexts.

4.2.2 *Code review reviewer assignment*

Rigby et al [124] analyzed the patch review process within the open source community via mailing lists of 5 large projects. Authors randomly sampled and manually analyzed email archives of Apache HTTP, Subversion, FreeBSD kernel, Linux kernel and KDE desktop environment. Typical review consists of two reviewers, who are long-standing committers in specific areas of the project. Paper describes both positive and negative reviewer attitudes and how a well structured patch helps to determine if change is worth reviewing. Ignored patches fail to generate interest within the core development team. Situations with too much involved reviewers usually lead to unproductive discussion.

Review Bot [10] is a reviewer recommendation tool based on review history of the source code. Each line in each file is inspected, and corresponding line reviewers are assigned points, with most recent reviews being scored higher. The reviewer recommendation list is created using summed points. The tool recommends reviewers who already participated in reviews of selected files. Consequently, it is unable to select reviewers for changes consisting only of new files. Additionally, Review Bot adds static analysts comments to review. The author also introduced the RevHistRECO algorithm, which uses file author history instead of review history as a baseline. Both algorithms were evaluated on VMware projects using accuracy at k, with Review Bot having better results.

Thongtanunam et al. [145] proposed the Revfinder tool to recommend reviewers utilizing previously reviewed file paths. This tool uses four string comparison functions to compute file path similarity between current review and past reviews: Longest Common Prefix, Longest Common Suffix, Longest Common Substring, and Longest Common Subsequence. The Borda count is used to create a single unified list of reviewers out of string comparison function results. Authors evaluated this tool against Review Bot [10] using *recall* and *Mean Reciprocal Rank* on Hamasaki dataset [45] expanded to include LibreOffice. The Revfinder outperformed Review Bot on all projects.

The authors of "Automatically Prioritizing Pull Requests" [151] propose PRioritizer tool. The tool examines open pull requests and sorts them using priority inbox approach. Pull requests are using a 1 day time window, within each window the machine learning model is trained to predict if the selected pull request will receive user action the next day. The authors evaluated Logistic Regression, Naïve Bayes and Random Forests using *precision* and accuracy on 475 GitHub projects, with Random Forests having best results. Additionally, they conducted a survey if the tool is useful among core contributors dataset projects, and received 12 positive feedback out of 21 responses total.

Hannebauer et al [46] provides a comparison of reviewer recommendation algorithms on four large open source projects (Firefox, AOSP, OpenStack, Qt). Authors evaluated File Path Similarity [144], Weighted Review Count, and six algorithms based on modification

expertise using Top-k accuracy. Algorithms based on review expertise yield better recommendations than those based on modification expertise (code authorship).

Yu et al. [167] propose to recommend pull request reviewers using both textual similarity of requests and social interactions between coders. The authors use a vector space model to represent text of pull request, and calculate similarity via cosine distance with already merged requests. In addition to text similarity, Yu et al. introduce project specific comment network graph, with graph nodes representing developers, and edges act of providing comments to pull requests. The weights of edges represent time decaying factor calculated as difference between relative dates of past comments timestamp and current pull request. The reviewers are selected through Breadth-First Search starting with pull request author. Both text similarity and graph travel results are combined to prepare a top-k reviewer list. In the follow-up paper [168], the authors evaluated the comment network approach against state-of-the-art file location based recommendation algorithm [145], cosine distance similarity recommendation and SVM used for multilabel classification. All algorithms were tested as standalone and with results combined with a comment network. The evaluation was done using *precision*, *recall* and *F1 score* on 5 large open source projects: rails written in Ruby, cocos2d-x in C++, ipython in Python, zf2 in PHP and netty in Java. The comment network obtained results similar to state-of-the-art file location recommendation as standalone method, with mixed approaches outperforming state-of-the-art.

4.2.3 *Code understandability*

Fowkes et al. [37] proposed a tool to automatically hide non-essential, less informative fragments of source code, such as Java "getter/setter" methods. This code summarization tool utilities topic model to detect fragments of AST. The authors conducted a developer study on 6 Java developers, which preferred automatic summarization results to raw source code.

Hindle et al. [51] seek to help software maintainers by introducing tags to project history. Tags are significant words from Latent Dirichlet Allocation topics, corresponding to non functional requirements.

Begel et al. [12] prepared graph data structures capturing interactions between developers and software artifacts within Microsoft corporation. Graph is used by two tools: "Hoozizat" and "Deep Intellisense" Visual Studio add in. The "Hoozizat" can be used for finding artifact ownership and code reuse outside teams. The "Deep Intellisense" provides a complete history of programmer activity such as code changes, creation of bug reports and forum activity related to specific code artifacts.

Antoniol et al. [3] proposed to detect features present in different applications, by discovery of feature microarchitectures. The method generates Abstract Object Language representation of the analyzed program and uses feature-relevant traces to detect code implementing a given feature. The authors conducted an evaluation on Mozilla family of web browsers and were able to detect save bookmark functionality across different programming languages.

Trockman et al. [148] analyzed results of Scalabrino et al[129] survey of 46 Java developers. The survey used source code of Spring Roo and Weka with questions if the developer understands the given code snippet. The authors analyzed features using Principal Component Analysis and found that no individual feature obtained from the survey was strongly correlated with understandability, and trained LASSO regression to identify

factors that correlate with it. The results show that the most important features for predicting understanding per developer are: textual coherence, is professional (binary value), is Java professional (binary value), number of method parameters, number of periods, methods internal documentation quality and max. line length.

4.3 PROBLEM STATEMENT

The code review system manages a list of commits to be reviewed, either in separate branches or as pull requests. Usually there are many potential reviewers for each review in industrial or open source projects. The code review system can present all available commits/pull requests to all candidates, but this approach has several disadvantages. Some commits might be omitted [151] or wait unnecessarily long, while the other changes are approved by people who do not possess required domain knowledge, resulting in defects or anti-patterns.

The reviewer recommendation system decreases the time of review, assigning reviewers according to the field of expertise of each candidate. The recommendation can be based on either human input, or on a self learning model, which examines the preferences of developers and their knowledge, based on history of reviews. In this chapter we focus on automatic reviewer recommendation systems.

In the system for each not yet reviewed pull request/commit and each prospective candidate a matching score is computed. This score is used to assess candidates, with higher value denoting better reviewer. After the changes are reviewed, the system includes the changes in the reviewer's past work.

An automatic reviewer section system, being the subject of our research, nominates reviewers for incoming commits using the information on the reviewers' past work and on the commit itself.

4.3.1 *Notation*

Let C be a stream of commits, ordered by time, corresponding to the version control repository branch. By C^t we denote the commit that appears at the time t in the commit stream C. Each commit consists of metadata and source code diff. We are interested only in the list of names of modified files. By $f(C^t) = [f_1^t, f_2^t, \dots, f_{l_t}^t]$ we will denote the list, where f_i^t is the name of the i-th file modified by the commit C^t for $i = 1, \dots, l_t$, where l_t is the number of files in the commit C^t. More precisely, f_i^t is a path, i.e. the string representing the file location in the operating system. This string will also be considered as a sequence of words separated by a special character depending on the underlying operating system. Additionally, let $R = [R_1, \dots, R_n]$ be the list of candidate reviewers.

Let $h(t)$ be the set of commits up to time t. Additionally, let us put $h_r(t)$ as the set of commits that at the time t have already been reviewed, and $h_r(t, R_j)$ as a set of commits that at the time t have already been reviewed by the reviewer R_j.

$$
\begin{aligned}
h(t) &:= \{C^i : i < t\} \\
h_r(t) &:= \{C^i : i < t \wedge C^i \text{ is already reviewed at } t\} \\
h_r(t, R_j) &:= \{C^i : i < t \wedge C^i \in h_r(t) \wedge C^i \text{ is reviewed by } R_j \text{ at time } t\}
\end{aligned}
$$

We assume that a commit is represented by the list of modified files, and that at any given time t for a reviewer R_k the set of his/her past reviews $h_r(t, R_k)$ is available.

Our goal is to construct a similarity function $s\colon C \times 2^C \to \mathbb{R}$. This function for a commit C^{t+1} and a candidate review history $h_r(t, R_i)$ quantifies how this commit matches this candidate's history. A larger value of $s\left(C^{t+1}, h_r\left(t, R_i\right)\right)$ indicates a better match.

The similarity function must be practically computable even for large repositories with abundant streams of incoming commits for the system to work efficiently.

4.4 THE PROPOSED METHOD

Each time new review arrives, the reviewer recommendation system has to designate the best reviewer candidates. Current state-of-the-art systems [10, 145] load and process the whole repository history. Note that, this approach is impractical, as it consumes too much system resources and time. For specific commit C^{t+1} we have to read and process $|h(t)| = t$ historical commits. Consequently, up to the time $t+1$ the reviewer recommendation system has to process $O(|h(t)|^2) = O(t^2)$ commits in total. Retrieval of all history data each time a new review arrives requires a worthwhile effort. Moreover, the required computation need is further multiplied by the number of present branches and repositories, and could become a serious issue for hosting companies like `GitHub` or `bitbucket`.

We propose a recommendation model based on profiles of reviewers as a means to solve the history retrieval problem. For each reviewer $R_i \in R$ we create his/her profile $P_i \in P$, where P is set of all available profiles. The profile P_i will be updated each time the reviewer R_i comments on a commit. Therefore, each time a new commit C^{t+1} is reviewed in the system, it will be added to the reviewer's profile.

Consequently, instead of the similarity function mentioned in Section 4.3.1, we rather use a similarity function $s\colon C \times P \to \mathbb{R}$ that for a commit C^{t+1} and a reviewer profile P_i quantifies how this commit matches this reviewer's history. In order to find the best candidate to review a commit C^{t+1} from a set of reviewers R, we compute the value of this function for each candidate profile. Therefore, we have to process $|P|$ reviewer's profiles but not t commits. At time $t+1$, such a system needs to process $O(|P| \cdot |h(t)|)$ commits. This is a significant improvement as number of potential reviewers $|P|$ is usually notably smaller than $|C|$. It holds that $|P| < 0.02 \cdot |C|$ for large open-source projects that we have evaluated, see Section 4.5 Table 4.4.

Therefore, to make the whole method feasible, we need (1) a data structure to store reviewers' profiles that has small memory footprint, (2) a suitable similarity function s that can be effectively computed, and (3) a fast profile updating mechanism to be applied whenever a reviewer comments on a commit. Fast updates of reviewer profiles are required as such system at time $t+1$ will have to perform $O(|h(t)|) = O(t)$ profile updates[1]. The remainder of this section is devoted to addressing those challenges.

4.4.1 *The reviewer profile*

In Section 4.3.1 we have introduced the notion of a commit as the list of modified files $f(C^t) = [f_1^t, f_2^t, \ldots, f_{l_t}^t]$, where l_t is the number of modified files in this commit. Each modified file path $f_i^t \in f(C^t)$ in a commit C^t is a string representing file location in the

1 This will be usually close to t as most often there is only one review per commit, see Table 4.4 in Section 4.5.

project. The file path consists of an ordered sequence of words separated by a special character. We omit the order of the sequence and treat it as a multiset of words, also known as a bag of words. We follow the semantics of multisets from [70, 134].

We denote the mapping that converts a file path f_i^t into the multiset of words (path segments) that occurs in f_i^t as $m(f_i^t)$. Let $m(C^t)$ be the multiset-theoretic union of $m(f_i^t)$ for all paths in $f(C^t)$:

$$m(C^t) = \bigcup_{f_i^t \in f(C^t)} m(f_i^t)$$

We will utilize $m(C^t)$ as the representation of the commit C^t used to construct the reviewers' profiles. We define the profile P_i for the reviewer R_i at the time t as the multiset-theoretic union of m for all commits previously reviewed by R_i, i.e.:

$$P_i(t) = \bigcup_{C^k \in h_r(t, R_i)} m(C^k)$$

In order to illustrate the above procedure, assume a repository containing commits C^1, C^2 and C^3 and an assignment of reviewers R_A and R_B as shown in Table 4.1.

Table 4.1: Example repository and assignment of reviewers.

Commit	List of reviewers	Paths (modified files)
C^1	R_A	"src/main/java/package1/SomeClass.java" "src/main/java/package2/DifferentClass.java" "src/test/java/package1/SomeClassTest.java"
C^2	R_A, R_B	"src/main/java/package2/DifferentClass.java" "src/test/java/package2/DifferentClassTest.java"
C^3	R_B	"src/main/java/package1/SomeClass.java" "src/main/java/package2/DifferentClass.java" "src/test/java/package1/SomeClassTest.java"

The commit C^1 has the following representation as the multiset $m(C^1)$:

$$m(C^1) = \left\{ \begin{array}{r} (\mathit{java}:3), (\mathit{main}:2), (\mathit{package1}:2), (\mathit{package2}:1), (\mathit{src}:3), \\ (\mathit{test}:1), (\mathit{SomeClass.java}:1), (\mathit{DifferentClass.java}:1), \\ (\mathit{someClassTest.java}:1) \end{array} \right\}$$

Since the commit C^1 is the first reviewed by R_A, $m(C^1)$ becomes the profile P_A of the reviewer R_A at time $t = 1$ as:

$$P_A(1) = \left\{ \begin{array}{r} (\mathit{java}:3), (\mathit{main}:2), (\mathit{package1}:2), (\mathit{package2}:1), (\mathit{src}:3), \\ (\mathit{test}:1), (\mathit{SomeClass.java}:1), (\mathit{DifferentClass.java}:1), \\ (\mathit{someClassTest.java}:1) \end{array} \right\}$$

After the commit C^2 (at the time $t = 2$) this reviewer's profile P_A becomes $m(C^1) \cup m(C^2)$:

$$P_A(2) = \left\{ \begin{array}{r} (\mathit{java}:5), (\mathit{main}:3), (\mathit{package1}:2), (\mathit{package2}:3), (\mathit{src}:5), \\ (\mathit{test}:2), (\mathit{SomeClass.java}:1), (\mathit{DifferentClass.java}:1), \\ (\mathit{someClassTest.java}:1) \end{array} \right\}.$$

4.4.2 *The implementation of the reviewers' profiles*

The proposed method is based on calculation of similarity between reviewers' profiles and reviewed commits. Consequently, the multiset data structures representing profiles must be efficiently implemented. Therefore, we have chosen hash tables, due to average constant time of both insertions and lookups. Each world (file path part) in a profile is mapped to its occurrence multiplicity via hash function. The hash tables are implemented in most popular programming languages. As a means to create the multiset-based representation $m(C^t)$ of a commit C^t, (1) all file paths modified in C^t are retrieved; (2) these paths are tokenized into words; and eventually (3) added to the hash table (by incrementing the occurrence counter). The calculation of $m(C^t)$ requires $|K|$ hash table lookups, where K is the number of words (path segments) in the $f(C^t)$ Note that the expected lookup time is $O(1)$, hence the complexity of calculation $m(C^t)$ is $O(K)$ The second most frequently performed operation in our method is an update of a reviewer's profile. Whenever a reviewer comments on a specific commit, the multiset representation of this commit is appended to his/her profile. Such operation is implemented as an iteration over items present in commit multiset. For each file path word present in commit, we increment the corresponding number of occurrences in the profile. The profile update requires $|S|$ hash lookups where S is the number of words in the multiset commit representation. Due to fact that the expected time of a hash lookup is $O(1)$, the time-complexity complexity of profile update is $O(S)$.

A reviewer profile can contain only the words that are path segments (names of directories and files) in the repository. Thus, a profile cannot be larger than a tree representation of all file paths in the directory structure of the repository. Table 4.8 in Section 4.5 shows memory footprint of such repository structures of analyzed projects. There are 47233 unique words for Android, 60521 for LibreOffice, 9759 for Open stack and 98897 for Qt. Therefore, the actual sizes of reviewer profiles are relatively small, making the whole algorithm applicable in practice.

4.4.3 *Computing the similarity between a profile and a pending commit*

For each new commit arriving in the repository, our method compares multiset representation of this commit and all reviewer profiles to grade review candidates. Various functions are generally used to measure the similarity between source code entities [143]. Most of those functions domains are not source code entities, hence the need of source transformation to fit the specific function domain.

We use two functions defined on sets, the Jaccard coefficient and the Tversky index.

The Jaccard coefficient is one of the most widely adopted similarity functions. It is defined for two sets X and Y as

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

It computes the fraction of overlapping items in the two sets.

The Tversky index is a generalized form of Tanimoto coefficient and is used to compare a variant to a prototype utilizing weights. For two sets X and Y it is defined as:

$$T(X, Y) = \frac{|X \cap Y|}{|X \cup Y| - \alpha|X - Y| - \beta|Y - X|}$$

where α corresponds to weight of a prototype and β corresponds to the weight of variant. The weights needs to be adjusted for specific application, for our method adjustment is described in Section 4.5.

We adapted both functions to work on multisets, by replacing set operations such as union, intersection and relative complement to those defined on multisets [70, 118, 134].

We decided to use the Tversky index as the main similarity function, because the importance ratio between profile and review can be adjusted. Additionally, we use Jaccard coefficient as a baseline multiset similarity function, in order to verify if our assumptions are correct.

Both functions can be efficiently implemented using hash tables. For the multisets X and Y of size M and N respectively, the calculation of union, difference and intersection requires at most $O(M+N)$ hash lookups which leads to the overall average time-complexity $O(M+N)$.

4.4.4 *The reviewer recommendation model*

In the proposed method, a new commit C^{t+1} waiting for review is mapped to its multiset-based representation $m(C^{t+1})$ and compared to profiles of reviewers P. To this end, the similarity score between $m(C^{t+1})$ and each $P_i \in P$ is calculated using similarity function s (currently either Tversky index or Jaccard coefficient). Table 4.2 summarizes the semantics of all necessary operations.

Table 4.2: Profile model operations

Function	Definition	Description
$P_i(t)$	$\bigcup_{C^k \in h_r(t,R_i)} m(C^k)$	Construction of a profile
$m(C^t)$	$\bigcup_{f_i^t \in f(C^t)} m(f_i^t)$	Creation of multiset-based representation
$s(C^{t+1}, P_i(t))$	$T(P_i(t), m(C^{t+1}))$	Tversky index as the similarity function
$s(C^{t+1}, P_i(t))$	$J(P_i(t), m(C^{t+1}))$	Jaccard coefficient as the similarity function
$top(C^{t+1}, n)$	$\langle R_{i1}, R_{i2}, \ldots, R_{in} \rangle$	Computation of top-n

We order reviewers according to obtained commit to profile similarity score (the highest first). Our assumption is that a higher score corresponds to better review proficiency for pending commits. Note that we calculate similarity for all profiles, hence we obtain the similarity score for all available reviewers. Consequently, we select only top n best fitting reviewers from ordered list. In case of several reviewers with the same similarity score, we break the ties using corresponding dates of the last review. A reviewer who did the most recent review takes precedence.

4.4.5 *Extensions to reviewer recommendation model*

Development of a software project, during its lifetime, is affected by team dynamics and the varying scope of each contributor's participation over time. Note that, some programmers might leave project, others increase their efforts, giving the most recent reviews more

importance. As an extension to the algorithm presented in the previous Subsection 4.4.4, we address this issue by introducing old review extinguishing in reviewer profiles.

Extinguishing is done by multiplying frequencies of words in profiles by an adjustable, lesser than one factor. Specific projects needs, such as focus of recent changes or enforcing involvement of first contributors can be obtained by adjusting the factor.

Although we were inspired by time-decaying Bloom filters [21], our approach to factor calculation and underlying data structure are unalike. Some authors researched evolutionary rules in the context of membrane computing, and used multiset extinguishing to decrease the factor of such rule activation [6].

The extinguishing factor for a number of days (or the number of commits in between) d is computed by the formula:

$$ex(d) = (\sqrt[l]{f})^d$$

where f is the decaying factor and l the number of days (commits) for which the decaying factor is to be fully employed. Let $id(C^t)$ and $date(C^t)$ denote respectively the sequence number of the commit C^t and the number of day between commits C^1 and C^t. Note that, if $f = \frac{1}{2}$ and $l = 180$, then after half a year (or after 180 commits) each item in a reviewer profile is halved. The adjustment of both parameters is discussed in Section 4.5.

As an alternative method to select top n candidates, we consider an approach not breaking ties for profiles with the same similarity score. In the *non tie-breaking* version of the algorithm we use the function

$$ntop(C^{t+1}, n) = \langle\langle R_{1,1}, \ldots, R_{1,m_1}\rangle, \ldots, \langle R_{n,1}, \ldots, R_{n,m_n}\rangle\rangle$$

, where all reviewers on a single sublist (e.g. $\langle R_{1,1}, \ldots, R_{1,m_1}\rangle$) have the same similarity to the commit. In particular, preceding lists contain reviewers with higher similarity to the commit than the consecutive ones.

Table 4.3: Profile model modified operations

Function	Definition	Description
$P_i^{id}(t)$	$\bigcup_{C^k \in h_r(t,R_i)} m(C^k) \cdot ex(id(C^k) - id(C^{k-1}))$	Construction of profile extinguished by id
$P_i^{date}(t)$	$\bigcup_{C^k \in h_r(t,R_i)} m(C^k) \cdot ex(date(C^k) - date(C^{k-1}))$	Construction of profile extinguished by date
$s(C^{t+1}, P_i^{id}(t))$	$T(P_i^{id}(t), m(C^{t+1}))$	Similarity extinguished by id
$s(C^{t+1}, P_i^{date}(t))$	$T(P_i^{date}(t), m(C^{t+1}))$	Similarity extinguished by date
$ntop(C^{t+1}, n)$	$\langle\langle R_{1,1}, \ldots, R_{1,m_1}\rangle, \ldots, \langle R_{n,1}, \ldots, R_{n,m_n}\rangle\rangle$	Non tie-breaking top n computation

4.4.6 *Possible extensions to profile creation*

Our method utilizes only a subset of the available features from project history, namely the reviewed file paths. Different features procured from additional views on the same

repository can be used to enhance the reviewer profiles. Either such features are already present in a given repository or can be computed by an additional algorithm. In particular, the authorship of each line belongs to the former category, and topic model of commit message to the later. Depending on selected features, pre-processing can also be adjusted to match the requirements of a specific project. Analysis of maintenance commits might cause unnecessary noise for some projects, but the other projects can benefit from it.

During development of a typical software project, the same developers constitute both reviewer and code authors groups. Therefore, authorship profiles can be prepared by aggregating code appropriate features. Consequently each developer would be assigned two distinct profiles, i.e. his/her reviewer profile and authorship profile.

Given these two profiles, we can consider at least the four following scenarios to use them: (1) recommendation via reviewer profile only, (2) recommendation via authorship profile only, (3) combining the similarity score from both profiles, (4) combining both profiles into one author/reviewer profile.

In order to combine scores to one unified list as in the third scenario, we would need to introduce different similarity functions for each kind of profile and additional post-processing of similarity scores. The fourth scenario can be implemented via using authorship information on file path level via introduction of those file paths to existing reviewer profiles. It seems easier to implement, on the other hand it lacks the flexibility of the third scenario.

Unfortunately we did not have a dataset with authorship information. Consequently, we implemented only the first scenario. However, we hope to analyze authorship features in the future work.

4.5 Evaluation results

In this section we detailed the empirical evaluation of the proposed method. In particular, we examined its accuracy, performance and memory footprint. We used the Thongtanunam dataset [145] that is based on mature and well-known open-source projects with long development history (see Table 4.4). We compared the obtained results with the state-of-the-art methods, i.e. ReviewBot [10] and Revfinder [145]. Moreover, we wanted to compute more quality metrics of their results than they had published. They showed only the *recall*, while we needed also the *precision* and *F-measure*. Unfortunately, the original implementation of Revfinder is not available. To this end we have reimplemented Revfinder, our implementation will be referenced as Revfinder*. Furthermore, we also wanted to assess the efficiency of their method that had not been mentioned in their article. We did not create a reimplementation of ReviewBot, since its quality had been verified to be inferior to Revfinder.

4.5.1 *The experimental setup*

We have conducted all experiments on a computer with two Six-Core AMD Opteron™ processors, 32 GB RAM, Intel® RAID Controller RS2BL040 set in RAID5, 4 drives Seagate Constellation ES ST2000NM0011 2000 GB. There was no other load on this machine during the experiments.

We measured time consumption using Python's *timeit* module. In particular, we took the average of 10 runs of our method on each project. In case of Revfinder* (for LibreOffice, OpenStack and Android), we also executed 10 runs and took their average execution time.

For Revfinder* on Qt we performed only one run, since the run-time exceeded 5 days and consumed excessive memory.

The memory footprint was measured using Python's *psutil* for the whole program. Additionally, selected data structures such as lists of reviewer profiles and arrays containing all distances were measured via Python *sys.getsizeof* function. Our sequential reimplementation of Revfinder had lower memory consumption than the parallel version. However, it was more time and CPU consuming.

4.5.2 *Experimental dataset*

We used the Thongtanunam dataset [145] that contains data on code reviews of the following projects: Android (a mobile operating system), LibreOffice (an office suite), OpenStack (a cloud computing platform) and Qt (an application framework). A summary of this dataset is presented in Table 4.4. This dataset contains contributors' activity recorded by the Gerrit code review systems. The features of each review are its unique identifier, the list of reviewers, the list of files, the creation date, the completion date, the name of the project and the information whether this code review has been merged.

Gerrit works with Git repositories. On default settings Gerrit allows a commit to be merged with the main repository only if this commit has not been blocked by any reviewer, and has at least one positive review.

Table 4.4: Statistics of processed projects

	Projects			
Statistic	Android	LibreOffice	OpenStack	Qt
Commits	5126	6523	6586	23810
(duplicated)	0	1	0	1
Reviewers	94	63	82	202
(only one review)	7	10	1	7
Reviews	5467	6593	9499	25487
First review	2010-07-13	2012-03-06	2011-07-18	2011-05-18
Last review	2012-01-27	2014-06-17	2012-05-30	2012-05-25
Average review distance				
seconds	9492	11032	4160	1354
ids	5.951814	1.499617	1.207865	1.140529
Sample size (MB)	3.8	5.5	4.2	19

Note that it is impossible to obtain 100% accurate reviewer recommendations, due to the fact that the authors of some reviews have no previous review history. In particular, the highest possible accuracy of recommendation are: 98% (Android) and 99% (LibreOffice, OpenStack and Qt).

Reviewers activity

In order to understand the Thongtanunam dataset [145] better, we analyzed the reviewers' activity.

The left column of Figure 4.1 shows the number of reviews per a single reviewer. For all four projects the diagram conforms to an exponential distribution. Most reviewers create less than 20 reviews for Android and LibreOffice and less than 60 reviews for OpenStack and Qt. In our opinion, these numbers result from a reviewer's focus on a specific bug or a feature request.

The middle column of Figure 4.1 shows the durations of individual reviewers' activity. In the case of Android and LibreOffice they are significantly longer than for Qt and OpenStack. It is probably caused by designated maintainers working for companies contributing to these projects.

The right column of Figure 4.1 depicts the durations of individual reviews. For all projects the majority of the reviews are completed up to three days for LibreOffice and OpenStack projects, up to two days for Qt and up to six days for Android. The longest review time and existence of outliers in case of Android suggests that a reviewer recommendation system can aid prioritizing the process [167].

4.5.3 *Applied metrics*

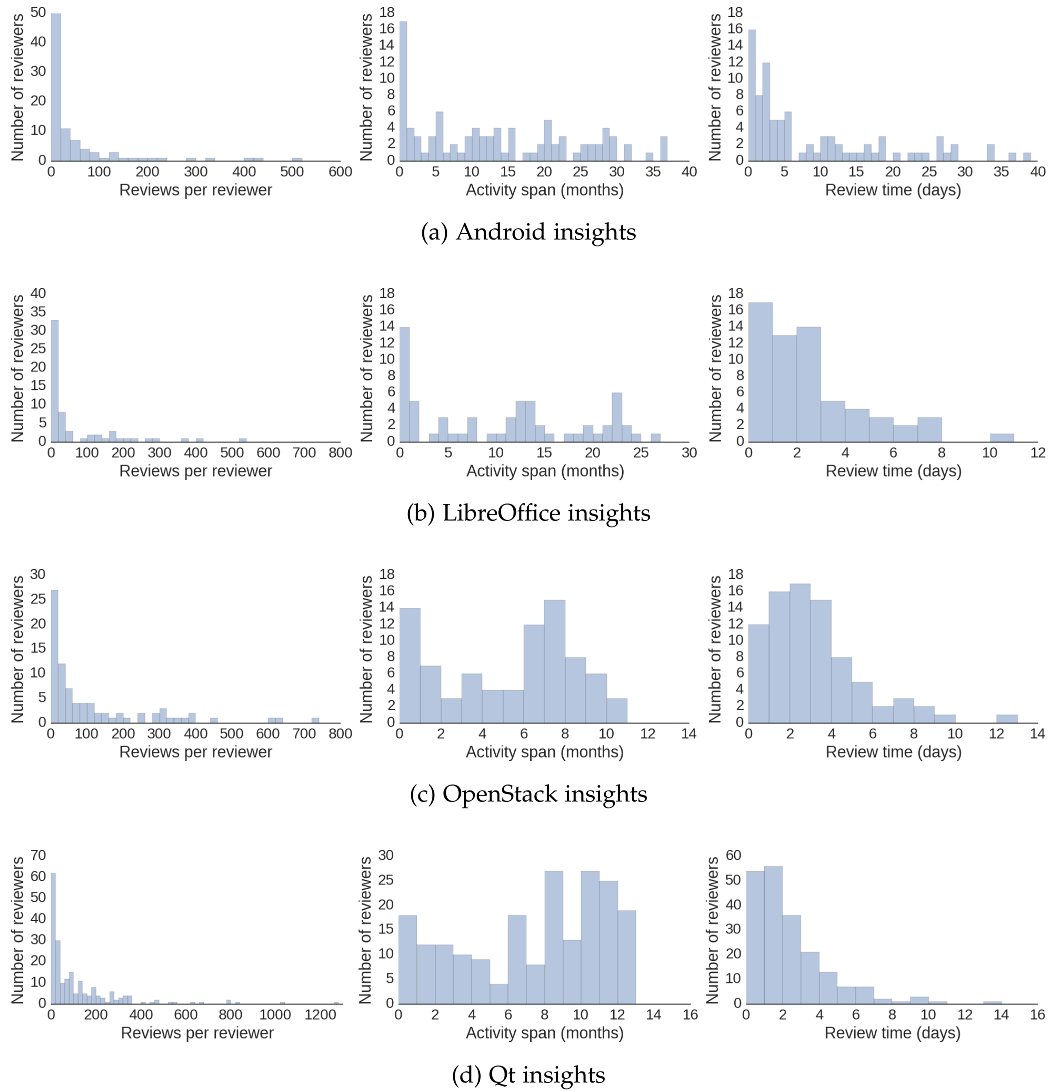
We computed the *precision*, *recall* and *F1 score* in order to assess the quality of our solution. We follow the metrics definitions as in Section 2.3.3.

The authors of Revfinder compared their tool to ReviewBot [10, 145] using only top-n *recall* and the *Mean Reciprocal Rank* (*MRR*). We compare their results to ours with respect to those metrics.

4.5.4 *Parameter selection*

As mentioned before in Section 4.4.3, the Tversky index requires two parameters α and β. Those parameters are importance weights of the multiset differences. Consequently, the α is applied to weight the difference between a profile and a commit, and β to the reverse difference We conducted an exhaustive search for both weights, under condition $\alpha + \beta = 1$, with values in range between 0 and 1, We used 10% of the dataset for this purpose, and found out that $\alpha = 0$ and $\beta = 1$ gave the best results. Therefore we utilize only the commit-to-profile difference, the intersection of a profile and a commit and their union, as the profile-to-commit difference is weighted 0.

In order to determine the best parameters for extensions of base algorithm (see Section 4.4.5), we further analyzed the dataset. We assumed that the impact of a review is halved after half a year. Thus, we put $l = 183, f = 0.5$ when extinguishing profiles' content by date. For the extinguishing of profiles' content by number of review, we utilized $l = 2500, f = 0.5$. The number 2500 is slightly less than half the number of reviews for Android, LibreOffice and OpenStack. Unfortunately, when using extinguishing we did not obtain significant improvement over base method. On the OpenStack and Qt projects, the extinguishing by number of reviews is able to obtain a higher *Mean Reciprocal Rank* than the base method. Intuitively, those projects have the larger number of reviews and smallest



(a) Android insights

(b) LibreOffice insights

(c) OpenStack insights

(d) Qt insights

Figure 4.1: Evaluated projects insights

average time between their submission (see Table 4.4), therefore older reviews are more likely to have significant impact.

4.5.5 *Recommendation system accuracy*

In our experiments we evaluated the method presented in Section 4.4 on the dataset described in Section 4.5.2 against state-of-the-art methods [10, 145].

We conducted experiments using four variants of our method and our replication of state-of-the-art [145], denoted as *Revfinder**. The differences in our methods variants lay in profile construction and used similarity functions. Three of them use the Tversky index. The first, denoted as *Tversky No Ext* utilizes standard profile construction, without extinguishing. The other two, denoted as *Tversky Ext id* and *Tversky Ext date*, use extinguishing by number of commits and number of days respectively. The last variant, denoted as *Jaccard* is based on Jaccard coefficient as the similarity function and standard profile construction mechanism.

In the tables we also present the results of *Revfinder* and *ReviewBot* as published in [10, 145] for reference.

Figure 4.2 presents our experimental results. The *Tversky No Ext* achieves better precision-to-recall ratio and higher *F-measure* compared to all other methods.

The results demonstrated in Table 4.5 show a detailed comparison of the methods listed above and reference state-of-the-art [10, 145]. Based on results obtained using top-n *recall* for $n \in \{1, 3, 5, 10\}$ and the *Mean Reciprocal Rank* we conclude that the *Tversky No Ext* outperforms other methods.

Consequently we have investigated if the observed improvements over state-of-the-art are statistically significant. Let $|top(C^{t+1}, n) \cap actual(C^{t+1}, n)|$ denote the ratio of successful recommendations to all recommendations., where $actual(C_{t+1}, n)$ is the actual n reviewers of the commit C_{t+1}. We calculated this ratio using results of top-n for all methods, all four projects and all $n \in \{1, 2, ..., 10\}$. Afterward we utilized the Levene test for equality of variances [83] to check that the variance of the results is not equal. Following that we employed the Kruskal-Wallis H-test [73] with null hypothesis "Median coefficients specified for each method are equal.", and p-value threshold equal to 0.05. Consequently we were able to reject this hypothesis for all tests. Next we used Student's t-test for independent samples with the hypothesis "Two method results have the same average values" with the same p-value threshold. We were able to reject this hypothesis for most pairs of results, namely 362 out of 400. The exceptions were (1) *Tversky No Ext* and *Revfinder** on OpenStack for $n \in \{6, 7, 8, 9, 10\}$, (2) *Tversky Ext id* and *Tversky Ext date* for Android and LibreOffice for all n.

Therefore we conclude that, in the majority of cases, our methods proposed in this chapter show statistically significant improvement over state-of-the-art for used metrics.

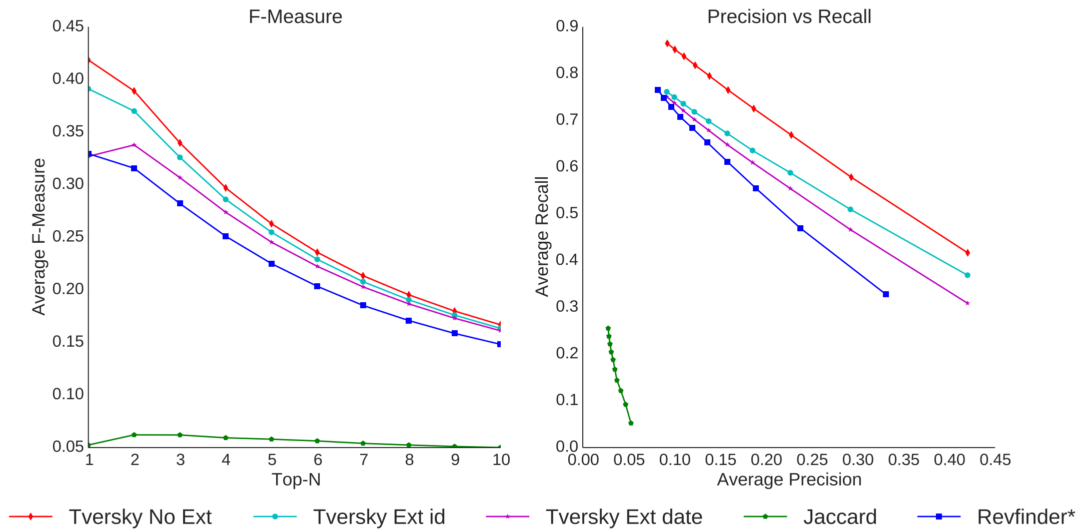


Figure 4.2: Metrics comparison

Non tie-breaking approach

The experiments presented above concern a tie-breaking version of the proposed model (see Section 4.4.5). However, the non-tie-breaking version is an interesting alternative. If

Table 4.5: The recall and MRR for all methods and four projects

	Method	Recall Top1	Recall Top3	Recall Top5	Recall Top10	MRR
Android	Tversky No Ext	0.5492	0.8034	0.8591	0.9066	0.7301
	Tversky Ext id	0.5138	0.7467	0.8004	0.8469	0.7290
	Tversky Ext date	0.4684	0.7315	0.7913	0.8460	0.6985
	Jaccard	0.0788	0.1859	0.2544	0.3798	0.4070
	Revfinder*	0.4686	0.7218	0.8098	0.8898	0.6640
	Revfinder	0.46	0.71	0.79	0.86	0.60
	ReviewBot	0.21	0.29	0.29	0.29	0.25
LibreOffice	Tversky No Ext	0.3353	0.5390	0.6571	0.8106	0.5799
	Tversky Ext id	0.3225	0.5216	0.6329	0.7872	0.5763
	Tversky Ext date	0.2692	0.4917	0.6097	0.7748	0.5340
	Jaccard	0.0239	0.0524	0.0747	0.1367	0.3559
	Revfinder*	0.2816	0.5183	0.6428	0.7972	0.5417
	Revfinder	0.24	0.47	0.59	0.74	0.40
	ReviewBot	0.06	0.09	0.09	0.10	0.07
OpenStack	Tversky No Ext	0.4177	0.6985	0.7967	0.8892	0.5500
	Tversky Ext id	0.2916	0.4829	0.5537	0.6157	0.5530
	Tversky Ext date	0.2260	0.4360	0.5176	0.6055	0.4924
	Jaccard	0.0835	0.1937	0.2609	0.3884	0.3931
	Revfinder*	0.3987	0.6846	0.7903	0.8854	0.5390
	Revfinder	0.38	0.66	0.77	0.87	0.55
	ReviewBot	0.23	0.35	0.39	0.41	0.30
Qt	Tversky No Ext	0.3655	0.6351	0.7480	0.8540	0.5973
	Tversky Ext id	0.3479	0.6014	0.7024	0.7962	0.6054
	Tversky Ext date	0.2720	0.5586	0.6746	0.7771	0.5500
	Jaccard	0.0226	0.0530	0.0785	0.1154	0.3926
	Revfinder*	0.1899	0.3403	0.4184	0.5356	0.5290
	Revfinder	0.2	0.34	0.41	0.69	0.31
	ReviewBot	0.19	0.26	0.27	0.28	0.22

we consider top-n groups of reviewers with the same score, we can observe the following. Figure 4.3 shows histograms of group size for non-tie-breaking *Tversky No Ext* top-1. For all projects singleton groups dominate. The number of smaller groups gradually decreases. This indicates that our methods usually find a small number of matching reviewers. Thus, a

valid question arises: Do we really need to break the ties? For large and long-term projects it may be fairly common that more than one person is responsible for specific parts of the project. Therefore, more than one reviewer has appropriate knowledge to perform a review. Under that assumption we investigate the predictive performance of this solution. We observe that top-n groups yield notably high *recall* rates. Table 4.6 shows the *recall* of top-n groups for several values of n. We conclude that this approach is also applicable, especially when a project's maintenance is naturally divided among working groups. It happens in e.g. subsystems in Linux kernel and FreeBSD operating system [124]. In such projects, having the high accuracy of group based recommendation may be desirable.

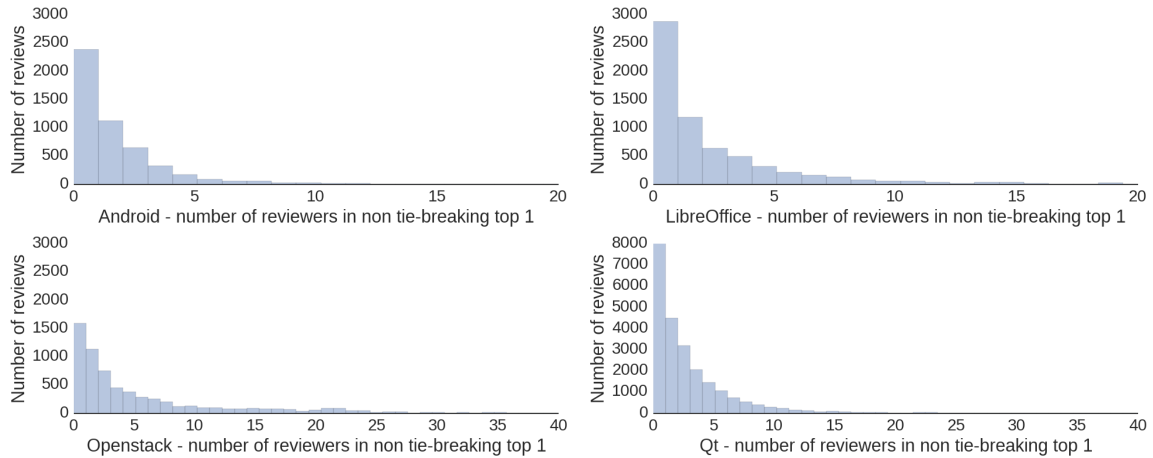


Figure 4.3: The number of recommended reviewers in Top-1 Tversky No Ext without tie-breaking. The horizontal axis shows the number of reviewers recommended. The vertical axis presents the number of commits for which this number of reviewers was recommended by this method.

Table 4.6: The recall and MRR of Tversky No Ext without tie-breaking.

	Recall				MRR
Project	Top1	Top3	Top5	Top10	
Android	0.7052	0.9058	0.9243	0.9317	0.8499
LibreOffice	0.4486	0.8013	0.8896	0.9402	0.6706
OpenStack	0.7486	0.9140	0.9417	0.9589	0.8289
Qt	0.6078	0.8798	0.9339	0.9633	0.7618

4.5.6 *Performance*

We measured the efficiency of three implementations, namely *Tversky No Ext*, *Jaccard* and *Revfinder**. We split the processing into three phases corresponding to operations in our model (see Table 4.2). They are the transformation of a commit into a multiset, the actual update of a profile and the similarity calculation in order to recommend the reviewers. Table 4.7 presents the results. The times for the three phases are total running times for all

operations per single project. The rightmost column presents average processing time per a review in a project.

The results presented in Table 4.7 are averages of running times for a number of executions. We executed 10 consecutive runs of the sequential version of our algorithm, both *Tversky No Ext* and *Jaccard*. In case of *Revfinder**, we used 10 consecutive runs only on Android, LibreOffice and OpenStack. We decided to evaluate *Revfinder** on Qt using only 1 run. This is due to the excessive consumption of resources for that project. One run took five days to compute. In all tests our method was faster by an order of magnitude. The coefficient of variation did not exceed 1% for all repeated runs. We used Mann-Whitney U two-sample test $n_1 = n_2 = 10, U = 100, p < 0.05$ with null hypothesis "Median execution time for *Tversky No Ext* and *Revfinder** are equal on the same project." on Android, LibreOffice and Openstack projects. Median times of 10 consecutive runs for *Tversky No Ext* on those projects were 248.84, 713.79 and 105.50 respectively. Median times of 10 consecutive runs for *Revfinder** on those projects were 15770.17, 24875.84 and 17235.29 respectively. We rejected the null hypothesis on all tested projects.

The memory footprint of profiles created by our algorithm is smaller than *Revfinder** (which uses 10 GB of RAM for Android, 12 GB for LibreOffice, 11 GB for OpenStack, and more than 32 GB for Qt). It is evidenced in Table 4.8.

This experimental observations confirm that our method uses significantly less computing power and memory than state-of-the-art methods. Thus, it is feasible to apply our methods in large repositories such as GitHub.

Table 4.7: Results of the performance evaluation

Project	Method	Total time (in seconds)			Average single review processing
		Multiset transformation	Profile update	Similarity calculation	
Android	Tversky No Ext	0.4232	5.3811	241.8630	0.0483
Android	Jaccard	0.4147	5.7651	132.3110	0.0270
Android	Revfinder*	-	0.0181	15797.9000	3.0819
LibreOffice	Tversky No Ext	0.6813	39.9458	668.6240	0.1087
LibreOffice	Jaccard	0.6741	39.9517	356.9950	0.0610
LibreOffice	Revfinder*	-	0.0249	24875.2000	3.8135
OpenStack	Tversky No Ext	0.3989	3.3080	101.1660	0.0159
OpenStack	Jaccard	0.3964	3.3920	56.6723	0.0092
OpenStack	Revfinder*	-	0.0254	17227.2000	2.6157
Qt	Tversky No Ext	2.3738	68.4784	4795.6800	0.2044
Qt	Jaccard	2.3844	68.2125	2651.9200	0.1143
Qt	Revfinder*	-	2.0000	432000.0000	18.1437

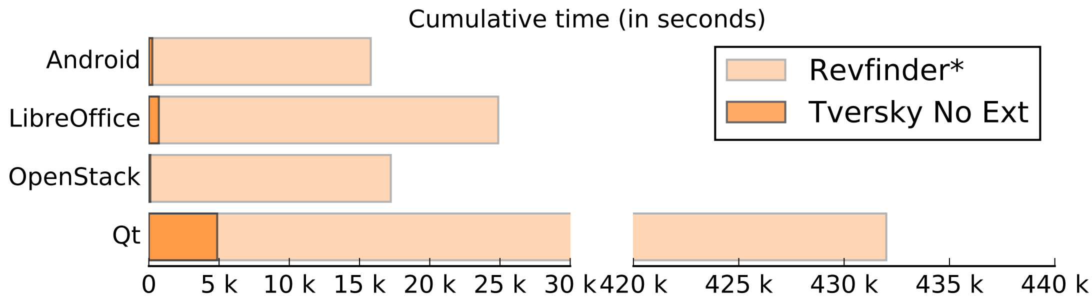


Figure 4.4: Performance difference between methods

Table 4.8: Memory footprint of Tversky No Ext (in MB)

		Multiset profile size	
Project	Virtual memory	All	Average
Android	396.6328	3.2235	0.0343
LibreOffice	411.6172	5.2938	0.0840
OpenStack	400.1172	1.8546	0.0226
Qt	530.0469	12.5634	0.0622

4.6 DISCUSSION

In this section we interpret our results and their significance when compared with state-of-the-art.

4.6.1 *The methodology*

We proposed a new algorithm solving the reviewer recommendation problem. To this end, we defined two distinct phases: (1) the aggregation of reviews into profiles, and (2) the calculation of profile-to-review similarity. A closer inspection of the typical repository reveals a disproportion of the number of occurrences between these two phases. Consequently, we decided to lower the cost of computing the second phase by doing more work in the first phase. Note that in the case of both state-of-the-art methods (Revfinder and Review Bot), the opposite is true. They do more processing in the second phase, and the first one is not utilized.

4.6.2 *Complexity comparison*

We accelerated the calculation of similarity in an analogous manner as a database index enables more efficient access to records. However, a database index adds storage and processing penalty. According to Amdahl law, improving performance of the most common and costly operation shall give us better improvement. Note that the number of review candidates is significantly lower than the number of commits/reviews. Thus the similarity

computation is going to be done significantly more often. Consequently it determines the complexity of the whole method. As shown in Table 4.7 the computation of similarity is more expensive than other phases. Therefore, it attests to our assumptions about the overall complexity.

4.6.3 *Empirical comparison*

The split to two distinct phases enabled us to evaluate different similarity functions, without the need to recompute all historical data. We tested Jaccard coefficient and the Tversky index with the goal of maximizing the similarity between aggregated profiles and reviews. This led to obtaining varying results and expanded our understanding of the dataset. By conducting experiments on adjustment of Tversky index weights we found that the most important feature for calculating similarity was the difference between multisets of a commit and a profile. Distinct aggregation phase allowed addition of optimizations, for example extinguishing of older reviews. We obtained statistically significant improvements for all metrics using Tversky index against state-of-the-art method Revfinder. Moreover, we were able to attain significantly lower time complexity. Although the extinguishing methods did not improve further our results, we have proven that those methods are comparable with the state-of-the-art (see Table 4.5).

4.6.4 *Profile construction*

We utilize only reviewed file paths as features for profile construction. The previous studies [145] had shown that such features can be used to obtain better results than code authorship. Furthermore, we were limited in our experiments due to the fact that the dataset does not contain detailed and complete authorship information. Note that it is possible to expand our model to work with authorship features, redefining profile construction and similarity usage as described in Subsection 4.4.6.

4.7 CONCLUDING REMARKS

Selection of appropriate reviewers in large projects is a difficult task. Moreover, an automatic reviewer recommendation system needs to work in an environment restricted by the available processing power and memory. In this chapter we introduced reviewers' profiles that aggregate their knowledge. We have shown a novel method to select code reviewers via usage of profiles. Our method achieved statistically significant advantage over state-of-the-art methods in terms of *precision*, *recall* and *F1 score*. In addition, the computational complexity of our method is also lower than the state-of-the-art. Intuitively, the presented experimental evaluation shows that profile based reviewer recommendation can be applied in large industrial systems.

5 ADAPTIVE BUG LOCALIZATION BASED ON BUG REPORTS

Contents

5.1 Introduction 45
5.2 Related work 46
5.3 Problem statement 49
5.4 Feature engineering 49
5.5 Proposed solution 51
5.6 Evaluation Results 56
5.7 Discussion 60
5.8 Concluding Remarks 65

5.1 INTRODUCTION

In this chapter we focus on bug localization using user reports. The bug localization can be seen as a specific form of information retrieval process, where we treat bug reports and software repository as queries and collection of documents respectively. In this setting files are ranked according to their relevance to specific bug report. Various models were proposed to automate this process [2, 22, 104, 109, 128, 131, 158, 165, 172].

Information about bugs and source code is present in multiple structured data sources, such as bug reports and repository change logs. Bug tracking systems and repositories are usually separate systems, only connected via messages present in commit metadata, containing identification number. Both of these resources are rich data sets for information retrieval purposes [62].

There are some challenges in the field of automated bug localization. One one hand, there exists a difference between the natural language used in bug reports and programming language employed in the source code. Therefore, utilization of simple lexical matching scores might cause suboptimal performance. To avoid this, a set of specialised, domain knowledge based features is often used [164, 165]. On the other hand, only a small fraction of files present in the project are buggy. Thus, both model, training data and used positive examples require careful selection. Consequently, one of the common shortcomings is the imbalance of positive examples and false positives generated by files closely related to bugs, such as files that are mentioned in the stack trace.

One of the earliest models in the field of bug localization checked presence of API calls in reported stack traces [2] or outliers in source code metrics [22, 104]. Current state-of-the-art methods benefit from information retrieval and machine learning algorithms, using features from multiple datasources. One source of such features is text data obtained from both bug reports and source code, with text similarity used to nominate suspect files [109, 172]. Abstract Syntax Tree (AST) of the compiled code is the other possible source, containing more fine-grained information, like names of classes, methods, variables, and source code comments [128]. Extracted stack-traces constitute another set of features [158]. Some

algorithms use composition of other methods, either utilizing linear combination of ranking scores [155, 156, 166] or learning to rank trained on a combined set of features [131, 165].

Ye et al. [165] presented notable results, preparing a new set of features and new learn-to-rank method. Furthermore, the authors proposed a new fine-grained dataset that better reflects practical applications than commonly used benchmark datasets [172]. Both method and dataset were an important step in this research field.

This chapter is based on a journal article under review [33]. Our contributions are as follows:

- we propose a novel, adaptive method to localize bugs;
- we publish an extension to the data set [165], that contains: intermediate data and final features, a variant of this data set that fixes missing bug report data, a corrected and enhanced feature, namely the API enriched lexical similarity, and last but not least the complete source code of our method;
- we provide a survey of existing state-of-the-art approaches.

5.2 RELATED WORK

We present several recent approaches to bug localization. The problem of bug localization has been thoroughly examined by numerous scientists. Their results are summarized in Table 5.1 and related to each other in Fig. 5.1. Below we elaborate on the state-of-the-art in detail.

Table 5.1: A summary of the related work on the bug localization. Top N and Accuracy@k metrics are exchangeable.

Name	Date	Dataset	Metrics
BugScout [109]	2011	ArgoUML, AspectJ, Eclipse, Jazz	accuracy, Top 10
BugCache [68, 120]	2011	Apache, Evolution, GIMP, Lucene, Nautilus	precision, recall, AUCEC
BugLocator [172]	2012	AspectJ, Eclipse, SWT, ZXing	Top N, MAP, MRR
BLUiR [128]	2013	*BugLocator dataset [172]*	Top N, MAP, MRR
two-phase [66, 109]	2013	8 modules from Firefox and Core in Mozilla	accuracy, precision, recall
Shivaji [133]	2013	11 projects	accuracy, precision, recall, f-score
BRTracer [158]	2014	AspectJ, Eclipse, SWT *(from BugLocator [172])*	Top N, MAP, MRR
AmaLgam [156]	2014	*BugLocator dataset [172]*	Top N, MAP, MRR
Ye et al. [164]	2014	AspectJ, BIRT, Eclipse, JDT, SWT, Tomcat	Top N, MAP, MRR
Zhang et al. [169]	2014	Eclipse, Equinox, JDT, Lucene, Mylyn	precision, recall, f-score, AUC
HyLoc [74]	2015	*Ye et al. dataset [164]*	Top N, MAP, MRR
AmaLgam+ [155]	2016	*BugLocator dataset [172]*	Top N, MAP, MRR
Ye et al.+ [165]	2016	*Ye et al. dataset [164]*	Top N, MAP, MRR
ConCodeSe [28]	2016	AspectJ, ArgoUML, Eclipse, SWT, Tomcat, ZXing	Top N, MAP, MRR
NP-CNN [55]	2016	AspectJ, Eclipse, JDT, PDE	Top N, MAP, AUC
BLIA [166]	2017	*BugLocator dataset [172]*	Top N, MAP, MRR
Shi et al. [131]	2018	Eclipse, SWT, ZXing *(from BugLocator [172])*	Top N, MAP, MRR

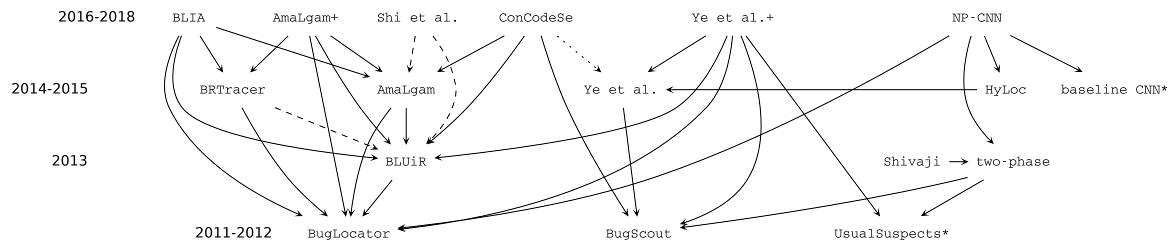


Figure 5.1: A graph representation of improvements in the related work. Solid arrows point from an improving work to the improved one, dashed arrows represent a partial improvement or similar results, while dotted arrows show an improvement but comparison for only a few projects. Some papers compare to naïve baseline methods that are marked with * on the plot. BRTracer [158] improves on BLUiR [128] except SWT project. Shi et al. [131] replicated the AmaLgam [156] and BLUiR [128], but were unable to obtain results reported with original papers. ConCodeSe [28] compared only Tomcat project with Ye et al. [164].

BugScout is a localization tool proposed by Nguyen et al. [109]. The tool is based on a modified Latent Dirichlet Allocation (LDA) topic modeling algorithm and Gibs Sampling method. BugScout uses two separate topic models, prepared for bug reports and source code files respectively. The models are connected utilizing links between bug reports and fixes. The algorithm estimates matching topic for new bug reports, and attempts to find files related to corresponding topics, using cosine distance as a similarity measure. BugScout was evaluated on 4 Java projects: Jazz, Eclipse, AspectJ and ArgoUML against SVM and LDA based approaches, with each project history split into 10 folds, using accuracy and Top 10 recommended files metrics. It outperformed other approaches, but was not tested against external algorithms.

Zhou et al. proposed BugLocator [172]. This localization model is based on the textual similarity between a bug report and the source code. The authors introduced a modification to the VSM model to not penalize larger source files via TF-IDF, as such files contain on average more bugs. They also prepared a dataset from bug reports of 4 open source projects: AspectJ, Eclipse, SWT and ZXing. An evaluation based on Top N file selection, *MAP* (Mean Average Precision) and *MRR* (Mean Reciprocal Rank) was done against VSM, LDA, SUM and LSI algorithms. BugLocator outperformed all of them.

BLUiR [128] was introduced as a bug localization algorithm finding similarity between AST entities and bug reports. The class names, method names, variable names and comments are obtained via AST parsing. This tool uses TF-IDF to select files that are most similar to the bug report summary and description. BLUiR was evaluated against BugLocator [172], using the same projects (AspectJ, Eclipse, SWT and ZXing) and metrics (Top N, *MAP*, *MRR*) and it achieved better results.

BRTracer [158] localizes bugs using segmentation of source code files and stack traces, with the goal to reduce the noise from larger files within the project. In this method the segment is created from one line of source file. The algorithm calculates the similarity between segments and bug reports using VSM, using only the segment most similar to bug report for further analysis. BRTracer utilizes an additional boost score calculated for files present on the stack trace. The authors evaluated BRTracer using Top N, *MAP*, and *MRR* metrics on 3 projects (AspectJ, Eclipse, SWT) against BugLocator [172] and BLUiR [128]. BRTracer outperformed BugLocator, and showed comparable results to BLUiR.

Ye et al. proposed bug localization using the Learning to Rank model [164]. The uses features based on text similarity measures on source code, including an enriched API hierarchy, class name similarity, collaborative filtering, bug fixing recency and frequency. The SVM^{Rank} package was used to provide the Learning to Rank model. The authors evaluated this method on 6 open source projects (AspectJ, Birt, Eclipse Platform UI, Eclipse JDT, Eclipse SWT, Tomcat) using *Accuracy@k*, *MAP* and *MRR* against BugScout [109] with better results.

Ye et al. further extended the base model [164] in following publication [165]. The extended version utilizes additional features obtained from AST parsing similarly to BLUiR [128], and retrieved from the class dependency graph. The method was evaluated using *Accuracy@k*, *MAP* and *MRR* against reimplemented BugLocator [172] and two baseline methods (Usual Suspects and VSM) and it outperformed BugLocator. The authors furhter tested their approach against BugScout [109], BLUiR [128], and BugLocator [172] on the datasets used in related publications, and got better results.

AmaLgam [156] bug localization tool combines BugLocator [172] and BLUiR [128] with repository data. This method utilizes weighted sum of other approaches scoring functions, with weights determined by experimental results per each project. The authors evaluated AmaLgam on the BugLocator dataset (AspectJ, Eclipse, SWT and ZXing) using Top N, *MAP* and *MRR* metrics against BugLocator and BLUiR and got better results. Wang et al. proposed an extended AmaLgam+ [155] by adding BRTracer [158] stack trace analysis as additional method, and changing weights preparation method to the genetic algorithm. AmaLgam+ obtained better results than AmaLgam [156], BRTracer [158], BLUiR [128] and BugLocator [172] on the same dataset.

The BLIA tool proposed by Youm et al. [166] localizes bugs on the file and method levels. The authors utilized the revision history, file contents, bug reports with comments and stack traces to find suspicious files. Methods present in each suspicious file are analyzed in terms of the similarity to bug reports. The similarity score between a file and a report is based on BLUiR [128]. The total weighted sum score is adapted from AmaLgam [156] with best weights obtained from experiments. The authors evaluated BLIA on the BugLocator dataset, using three projects (Aspectj, SWT, ZXing) with Top N, *MAP* and *MRR* metrics. BLIA got better results than AmaLgam [156], BugLocator [172], BLUiR [128] and BRTracer [158].

ConCodeSe [28] aims at improving the bug localization without depending on features from project history. The tool is based on a probability score and a lexical score implemented in Apache Lucene search engine. The authors customized Lucene's Standard-Analyzer to tokenize bug reports. ConCodeSe was evaluated against BugScout [109], BLUiR [128], Ye et al. learning to rank [164], and AmaLgam [156] using Top N, *MAP*, *MRR* on AspectJ, Eclipse, SWT, ZXing, Tomcat and ArgoUML. The tool is able to outperform BugLocator [172] on all projects, AmaLgam [156] on four projects, BLUiR [128] on three projects and Learning to rank [164] on one project (Tomcat).

Shi et al. [131] tested various learning to rank models, to find the one best suited for bug localization. The authors used algorithms implemented in the RankLib [27] library: Random Forests, MART, RankNet, RankBoost, LambdaMart, ListNet, AdaRank, CoordinateAscent. The features used were similarity scores between bug report summaries/descriptions with class names, methods, variables and comments, stack traces, version history per each file, dependence graphs, and report to report similarities. The best tested learning to rank method, CoordinateAscent, was evaluated against AmaLgam and BLUiR [128], using 3

open source projects (Eclipse, SWT, ZXing) from BugLocator dataset [172] and Top N, *MAP*, *MRR* metrics with better results on Eclipse and SWT.

The two-phase method proposed by Kim [66] aims to find files relevant to a specific bug report by training Naïve Bayes on a bag-of-word features obtained from previous bugs. The authors use the probability per each file to select top k files related to bug report. In order to enhance the performance, a separate grading classifier was trained on already solved bug reports. This is used to filter out bug reports without enough information provided to predict files. The method was evaluated against Usual Suspects and BugScout [109] in terms of likelihood (accuracy), precision and recall. It outperformed both approaches with statistical significance.

Huo et al. [55] propose NP-CNN method based on Convolutional Neural Network to locate buggy files, with the goal of utilizing program structure and natural language processing simultaneously. The authors define an additional cost function for misclassification due to the unbalanced nature of defect to non-defect ratio. CNN was evaluated on 4 open source project: AspectJ, Eclipse JDT, Eclipse Platform and Eclipse PDE against BugLocator [172], the two-phase method [66], HyLoc [74] and baseline CNN without adjustments for programming language structure encoding, with metrics like Area Under Curve (AUC), *MAP* and Top N. NP-CNN outperformed previous models, with statistical significance on all projects, except JDT, where it was the second best after the other method based on CNN.

HyLoc [74] is based on combining deep neural networks with textual similarity models. It uses a neural network to find pairs of related terms between reports and source code files. HyLoc was evaluated on the Ye et al dataset [164] using Top N, *MAP*, *MRR* metrics against BugLocator [172], Ye et al. [164] and Kim et al. [66] with better results on all projects.

5.3 Problem statement

Typical bug fixing process requires a software user, who encounters a bug to report it to the bug tracking system. After that the developer who chooses or is chosen to fix the bug needs to localize appropriate source files within the project and modify them.

5.3.1 *Notation*

Let B be a stream of bug reports, ordered by time. By b^t we denote the bug report fixed at time t. Each bug report contains the title text and description text. Let C be a stream of commits, ordered by time, corresponding to the version control repository branch. Let c^{t-1} be the state of project before applying fix for b^t, corresponding to specific commit. Let S^t be the set of files present in repository for commit c^{t-1}, and S set of all file states. Let s_i^t be one source file in S^t.

Our goal is to construct a ranking function $r\colon B \times S \to \mathbb{R}$. This function for a bug report b^t and file s_i^t ranks how this file is related to a bug report. The ranking function must be practically computable even for large repositories.

5.4 Feature engineering

Bug localization models operate on a set of features that captures relationships between files and bug reports. A large number of features was proposed over time (see e.g. [164,

165] and references therein). Each pair of a bug report and a source file (b, s) is represented as a vector of k features: $\breve{}(b, s) = [\phi_i(b, s)]_{1 \leqslant i \leqslant k}$. We use the same set of 19 features as the authors of [165], following their naming and numbering, as listed in Table 5.2, except ϕ_2, which we replaced. We slightly adjusted formal definitions of these features to make it simpler and more accurately reflect their meaning. The features are normalized using standard min–max scaler $\frac{\phi_i - \min_n \phi_i}{\max_n \phi_i - \min_n \phi_i}$, with $\phi_i(b, s)$ values from current fold n that are below or above values from the previous fold, that is $\min_n \phi_i$ and $\max_n \phi_i$, set to 0 or 1, respectively.

Table 5.2: Features used in ranking model, proposed by Ye et al. [164, 165]. *sim* is the cosine similarity. Slight notation changes for features ϕ_2, ϕ_3, ϕ_5 are marked in bold. We use ϕ_2^* feature instead of ϕ_2, see Section 5.4 for rationale. *Query dependent* features are those that depend on both the bug report r and the source code s.

Feature	Description	Formula	Query dep?
ϕ_1	Surface lexical similarity	$\phi_1(b, s) = \max(\{sim(b, s)\} \cup \{sim(b, m) \mid m \in s\})$	Yes
$\boldsymbol{\phi_2}$	API-enriched lexical similarity	$\phi_2(b, s) = \max(\{sim(b, s.api)\} \cup \{sim(b, m.api) \mid m \in s\})$	Yes
$\boldsymbol{\phi_3}$	Collaborative filtering score	$\phi_3(b, s) = sim(b, \text{concat}(\{b.summary \mid b \in br(b, s)\}))$	Yes
ϕ_4	Class name similarity	$\phi_4(b, s) = \lvert s.main_class \rvert \cdot \mathbb{1}[s.main_class \in s.summary]$	Yes
$\boldsymbol{\phi_5}$	Bug-fixing recency	$\phi_5(b, s) = \left((b.date - last(b, s).date).months + 1\right)^{-1}$	Yes (Timestamp)
ϕ_6	Bug-fixing frequency	$\phi_6(b, s) = \lvert br(b, s) \rvert$	Yes (Timestamp)
ϕ_7	Summary–class names similarity	$\phi_7(b, s) = sim(b.summary, s.class)$	Yes
ϕ_8	Summary–method names similarity	$\phi_8(b, s) = sim(b.summary, s.method)$	Yes
ϕ_9	Summary–variable names similarity	$\phi_9(b, s) = sim(b.summary, s.variable)$	Yes
ϕ_{10}	Summary–comments similarity	$\phi_{10}(b, s) = sim(b.summary, s.comment)$	Yes
ϕ_{11}	Description–class names similarity	$\phi_{11}(b, s) = sim(b.description, s.class)$	Yes
ϕ_{12}	Description–method names similarity	$\phi_{12}(b, s) = sim(b.description, s.method)$	Yes
ϕ_{13}	Description–variable names similarity	$\phi_{13}(b, s) = sim(b.description, s.variable)$	Yes
ϕ_{14}	Description–comments similarity	$\phi_{14}(b, s) = sim(b.description, s.comment)$	Yes
ϕ_{15}	In-links = # of file dependencies of s	$\phi_{15}(b, s) = s.inLinks$	No
ϕ_{16}	Out-links = # of files that depend on s	$\phi_{16}(b, s) = s.outLinks$	No
ϕ_{17}	PageRank score	$\phi_{17}(b, s) = PageRank(s)$	No
ϕ_{18}	Authority score (HITS)	$\phi_{18}(b, s) = \text{auth}(s)$	No
ϕ_{19}	Hub score (HITS)	$\phi_{19}(b, s) = \text{hub}(s)$	No
$\boldsymbol{\phi_2^*}$	full API–enriched lexical similarity	$\phi_2^*(b, s) = \max(\{sim(b, s.api^*)\} \cup \{sim(b, m.api^*) \mid m \in s\})$	Yes

We reimplement Ye et al. [164, 165] feature extraction using Python and Java (for `ASTParser`). To construct the features we used Pandas [97], Sklearn [117], Numpy [110], NetworkX [43], Natural Language Toolkit (NLTK) [92] and Eclipse JDT ASTParser. The file dependency graph was created by parsing all Java source files using the Eclipse JDT ASTParser and inferring the file dependencies.

Main differences compared to [165] are new ϕ_2^* feature and other TF-IDF weight scheme[1]. We publish feature engineering code and the values of resulting features as a supplement to the dataset [164, 165].

New feature ϕ_2^* rationale

The original ϕ_2 feature proposed in [164, 165] is constructed based on API description extracted from the project documentation. While this feature can enhance the overall process of bug localization, the way it is constructed in [164, 165] poses two problems. First, it may

1 Read Section 5.6 to see how this impacts the results.

leak information as it is based on snapshots of documentation, thus some bug reports may be evaluated against the documentation not available at the time of the report. Based on the API URLs present in the published dataset, the documentation was gathered from Indigo (2011) and Kepler (2013) versions of Eclipse projects. The earliest bug reports are from 2001. Second, it does not contain all entries available in the documentation. For instance the main class for plugin development UIPlugin from Eclipse Platform UI is not included in the snapshot.

We propose new feature ϕ_2^* that solves mentioned problems. This feature is based purely on the API derived from AST of the source code available before the bug report was reported, $s.api^*$. More computational effort is required compared to original ϕ_2, but it does not leak information, and is not subjective to the initial API selection.

Text processing differences

Small differences exist between the paper [164] and the feature extraction code we received from its authors. Implementation uses WordNet lemmatizer, and manually adjusted stop words per each project which was not described in the paper [164]. We follow procedure from the publication [164].

As for the TF-IDF weighting scheme, we decided to use an established scheme used in the scikit-learn library[117] and in the Apache Lucene search engine. This scheme differs from Equation 1 by defining:

$$idf(t, D) = \log \frac{1 + |D|}{1 + |\{d : t \in d\}|} + 1,$$

where t is term and D set of all documents.

5.5 PROPOSED SOLUTION

For each new bug report, our method localizes related files by calculating likelihood score per each file present in the repository at the moment of bug report creation. Files more probable to be the cause of the bug will get the higher score than others.

We train the algorithm on the dataset containing both bug reports, and related commits fixing them. Our assumption is that there is some history of project development and bug reporting present in the repository and bug tracker. We consider it as an important aspect, therefore, we designed the model so it could adapt to the project over time. Similarly to other approaches we treat ordered bug reports as a time series with a sliding window [165]. This window splits reports into folds of constant size. Due to ongoing development intertwined with bug fixing commits and changes by participating developers, each fold can have different characteristics. Consequently, we train our method on fold n to localize bugs for fold $n+1$. Thus, we adapt our model to the latest fold characteristics, rather than utilizing the characteristics of a bug report, like in[103].

Note that, our goal is to find the best method and parameters, without usage of any features not present in either bug tracker or repository. Therefore we set all parameters automatically, without the need for separate parameter fitting stage [165] or manual parameter fitting [128, 172]. If some weights are to be established, we avoid setting them manually, and have the algorithm adjust them. Given a bug report, the algorithm performs the bug localization by computing the likelihood score for each file present in the repository at the

time of the bug report creation. Then, it uses this score to present files more likely to be a cause of the bug. More probable files will get a higher score.

Using learning to rank approach in the context of bug localization requires a special setup. The main challenge is the huge imbalance between relevant and irrelevant files for a given bug report. For example, 4 largest projects from dataset[165] have between 4000 and 6000 files, while the median of relevant files for a bug report is between 1 and 3. Therefore, a certain data preparation is required in order to mitigate the imbalance problem.

Based on related works we generalized the typical setup used in this case (see Fig. 5.2 and Table 5.3).

Figure 5.2: The block schema of a general learning to rank approach in bug localization.

The *initial ranking* selection is a preliminary step used either in the *imbalance handling* method or for establishing the *training target*. The examples of such *initial rankings* are: the feature ϕ_1 in Ye et al.+ [165], results of BLUiR [128] algorithm used in Shi et al. [131], and a random order for the genetic algorithm AmaLgam+ [155]. We propose a custom approach, described in Section 5.5.1, where we use a score based on (per–fold) adaptive feature weights approach. Then, there is the *imbalance handling* method which cuts off most of the irrelevant files making the training set more balanced. In [131, 165] all but top 200 of irrelevant files based on the *initial ranking* are excluded from the training set. In AmaLgam+ [155] there is no imbalance handling. We use a two step approach where we first cut using the feature ϕ_2 as the ranking and use the top 200 irrelevant files. Then we proceed with an adaptive cut-off which utilizes the *initial ranking* to further reduce the number of files. Finally, there is the *training target*, that will be used in the training process. The authors of AmaLgam+ [155] maximize the function $e^{MAP+MRR}$ on results of a genetic algorithm selecting weights on randomly selected 5% of the dataset. Shi et al. maximize the MAP metric [131]. In [165] a simple training target is used, based on a binary relation ie. $+1$ for relevant files and -1 for irrelevant files. On the other hand, we use more sophisticated mechanism based on an ameliorated score. This leads to a fine-grained ranking that induces a linear order.

5.5.1 *Initial ranking*

This method is based on constructing the relevancy score per each file, to be used for point-wise learn-to-rank training target construction. We were inspired by feature importance analysis.

To create ranking we propose several scoring functions, consisting of various statistical tests, classification methods or information theory measures. Each one of those functions is linear combination of features $p_i(b, s) = \sum_k w_k \cdot \phi_k(b, s)$, where weights w_i are set heuristically, based on several proposed criteria. Scoring functions are evaluated on each

Table 5.3: Comparison of selected learning to rank bug localization methods. In learning to rank all refers to pointwise, pairwise, listwise.

	Ye et al.+ [165]	Shi et al. [131]	AmaLgam+ [155]	**Proposed method**
Initial ranking ($\preceq_{IR}$)	feature ϕ_1	BLUiR [128]	random	ϕ_2^* **and adaptive scoring**, Section 5.5.1
Training target	relevant files: +1, irrelevant files: −1	max MAP	max $e^{MAP+MRR}$	**ameliorated score**, Section 5.5.2
Imbalance handling	top 200 irrelevant files based on $\preceq_{IR}$		none described	**adaptive cut-off of irrelevant files**, Section 5.5.3 Initial cut: top 200 irrelevant files based on feature ϕ_2^*, proceeded by Adaptive Cut: adaptive bottom % of selected files based on $\preceq_{IR}$.
Learning to Rank	SVMrank [60] pairwise	Coordinate Ascent RankLib [27] listwise	JGAP [98] listwise	**adaptive SGD regression**, Section 5.5.4 pointwise

training sample for the fold, which is the same as the training subset for the fold in Ye et al. [165].

Weights heuristics are based on estimation of how well each feature ϕ_i can distinguish between relevant and irrelevant files.

Our intuition is that weights are to be used to rank each feature importance akin to how they are used for feature selection [42]. We present all used functions and weights in Table 5.4 Weights are scaled by sum of all features ϕ_i for given scoring criteria.

We based the first group of heuristics on statistical tests. In particular we split each feature into relevant and irrelevant groups and then use relevant statistics as a measure of corresponding feature importance. That said we utilize Kruskal-Wallis H-test [73] to test whether the population median are equal, the T-test for independent samples [107] to check if means are significantly different, chi-square test [107] to find out features independent of relevancy class, and Levene test for equality of variances [82] to assess the equality of variances between relevant and irrelevant groups.

In the second group we have various classification methods that are used for their feature importance estimators. Each classifier is trained to predict fix and non-fix files. We use AdaBoost tree classifier [38, 49], an Extremely Randomized Trees classifier [40] and Gradient Boosting regression [50] to obtain feature importance estimates.

In the last group, we have heuristics based on the index of dispersion [107], maximum absolute deviation [107], and mutual information [125].

The use of index of dispersions is motivated by the idea that better distinguishing features have it different for relevant and irrelevant files. We also check the maximum absolute deviation variances calculated on relevant files features to capture the characteristics specific to only those files. Mutual information is used to check for the most discriminating features in the context of the target variable.

The simplest heuristic is based on a constant set of weights, where $w_i = 0.5$.

Table 5.4: Scoring functions and weight schemata used in the adaptive scoring phase, where ϕ_i - all values for feature i, ϕ_i^{fix} - values for feature i restricted to files used in fix, ϕ_i^{irr} - values for feature i restricted to irrelevant files, Y - true/false values for each file, true if file is a fix, else false

Based on	**Weights**
W statistics from Levene test for equality of variances with median as center function [82]	$w_i = W_{\phi_i} / \sum_{j=1}^{n} W_{\phi_j}$
H statistics from Kruskal-Wallis H-test [73]	$w_i = H_{\phi_i} / \sum_{j=1}^{n} H_{\phi_j}$
T statistics from T-test for independent samples [107]	$w_i = T_{\phi_i} / \sum_{j=1}^{n} T_{\phi_j}$
χ^2 statistics from chi-square test [107]	$w_i = \chi_{\phi_i} / \sum_{j=1}^{n} \chi_{\phi_j}$
Features weights computed by AdaBoost SAMME.R Classifier [38, 49]	model specific
Features weights computed by Extremely randomized trees classifier[40]	model specific
Features weights computed by Gradient Boosting regression [50]	model specific
Mutual Information between Discrete and Continuous Data Sets [125] denoted as I	$w_i = I(\phi_i, Y) / \sum_{j=1}^{n} I(\phi_j, Y)$
Index of dispersion [107]	$w_i = D_{\phi_i^{fix}} / D_{\phi_i^{irr}}$ where $D_{\phi_i} = \sigma^2_{\phi_i} / \mu_{\phi_i}$
Maximum absolute deviation variances [107]	$w_i = \text{mean}\left(\lvert\phi_i^{fix} - \max \phi_i^{fix}\rvert\right)$
Predefined set of weights	$w_i = 0.5$

Adaptation: selection of the best scoring function

In order to select the best set of normalized feature weights w_i (see Fig. 5.2), we evaluate weights from each scoring function under *Mean Average Precision (MAP)* metric, defined as in Section 2.3.3. In particular, we utilize two-way cross validation for this purpose. Each training fold of 500 bug reports is split into two equal sized disjoint datasets. We use each one of those datasets to find the weights for each scoring function, and the corresponding other to compute metrics. Consequently we obtain the metrics on the whole training fold, and utilize them to select the best weights and scoring function. To create one set of weights for the training fold we combine best scoring weights from two cross-validation datasets.

As mentioned before, the process is repeated per each training fold, allowing for various models to be selected. We decided to use adaptation as training folds tend to change their characteristics. Note that the scoring function p can be used as standalone ranking function r.

5.5.2 *Training target: ameliorated score*

Based on Shi et.al. [131] findings usage of pointwise learning-to-rank setup to bug localization can result in suboptimal performance, when utilizing coarse–grained binary relation as a training target. Because of that, it is essential that the training target is fine-grained and linearly ordered.

Our training target is based on the following ameliorated score function. We add the maximum of feature values to the score of relevant files.

$$p^*(b, s) = \sum_{i=1}^{n} w_i \cdot \phi_i(b, s) + \mathbb{1}_{[s \in \mathit{fixes}(r)]} \cdot \max_{i=1..n} \phi_i(b, s),$$

where $\mathit{fixes}(b)$ is a set of fixed files for a given bug report b. This modification establish proper training order, by ensuring that all fixed files have greater p^* score than non–fixes. Both p and p^* are fine-grained and of linear order for files under a given bug report.

5.5.3 *Imbalance handling: adaptive cut–off*

The files not related to a bug report severely outnumber the buggy files. To be able to correctly train regression models we need to create a training set with appropriate ratio of bug related and unrelated files (i.e. all files with bugs, and small sample of other files). We want the sizes of created classes to be of similar magnitude. First we narrow the files by taking top 200 irrelevant files based on the ϕ_2^* feature, similarly to [131, 165].

We narrow the training set with the cutoff function,

$$t_f(b) = \mathit{fixes}(b) \cup \{s \notin \mathit{fixes}(b) \mid 0 < p^*(b, s) \leqslant c_f(b)\},$$

which selects non-defect (irrelevant) files for the training set. It takes source files with the lowest score per bug report, up to a given fraction of f% of the number of defect (relevant) files. The cutoff value is defined as

$$c_f(b) = \max\left\{ p^*(b, s) \mid s \in P_{|\mathit{fixes}(b)| * f} \right\},$$

where P_n be the set of n smallest elements of set of scores P, that is the maximum score in the set smallest scores with the cardinality of f% of the set of relevant files. We evaluate each variation of regression models with a cutoff function using 5%, 10%, 15%, 20%, 25% and 30% of previously chosen irrelevant files per bug report.

5.5.4 *Pointwise learning to rank method*

The starting point for this method are scores generated by selected best performing scoring method $p(b, s)$. Having an initial score enables us to train pointwise learning-to-rank algorithms. The goal is to train a regression model based on an ameliorated scoring of the training dataset.

Likewise to the selection of scoring weights, we evaluate several regression models. In particular we use variants of Stochastic Gradient Descent regression [171], with following loss functions: ordinary least squares [171], Huber [113], epsilon insensitive [137] and squared epsilon insensitive [137] and no regularization or regularization via L1 norm, L2 norm, or Elastic Net.

Adaptation: selection of the best model

Similarly to scoring, the regression model is also trained and evaluated using 2-way cross validation. The models search space consists of Cartesian product of all possible variants of training models and cutoff thresholds. We train each model after applying the cutoff function on training data. The winning model is selected based on *Mean Average Precision (MAP)* of bug files that it selects.

5.6 EVALUATION RESULTS

In this section we detailed the empirical evaluation of the proposed adaptive methods. We compare the state-of-the-art [165] results to ours on the same fine grained dataset [165]. Note that most of related work utilizes older dataset, introduced by authors of Buglocator method [172], as shown in Section 5.2. For completeness we include the comparison using the Buglocator dataset. Furthermore, we assess the performance of our proposal.

Table 5.5: Publicly available benchmark datasets used in this chapter.

Dataset	Project	# of bugs	appr. # files	Missing desc.
Ye et al.	AspectJ	593	4439	21%
	BIRT	4178	6841	28%
	Eclipse UI	6495	3454	19%
	JDT	6274	8184	15%
	SWT	4151	2056	21%
	Tomcat	1056	1552	50%
BugLocator	AspectJ	286	6485	
	Eclipse 3.1	3075	12863	
	SWT 3.1	98	484	
	ZXing	20	391	

5.6.1 *Applied metrics*

We computed the *Accuracy@k*, the *Mean Average Precision* and *Mean Reciprocal Rank* order to assess the quality of our solution, following the metrics definitions as in Section 2.3.3. Consequently, we compare the state-of-the-art and previous works results to ours with respect to those metrics.

5.6.2 *Fine grained dataset*

This dataset was proposed by Ye et. al. [164, 165], and is publicly accessible[2]. It contains six open-source Java projects: AspectJ, BIRT (Business Intelligence and Reporting Tools), Eclipse Platform UI, JDT (Java Development Toolkit), SWT (Standard Widget Toolkit), and Tomcat. All of these projects use Bugzilla as the bug tracking system and Git as the version control system. This dataset closely resembles real life use cases; thus it should be preferred over commonly used BugLocator dataset [172].

The connection between bug report and commit was created by searching commit messages for special phrases such as "bug 31946" or "fix for 31946" according to the heuristics proposed in [26]. Only closed bug reports with clear corresponding fixed files were considered. As in [164, 165], the version of the corresponding software package just right before fix was used as the substitute of the exact version for which the bug was reported.

During investigation of the dataset we discovered that some of the bug reports do not contain the description, while it is present in the related Bugzilla. The percentages of missing descriptions per project are presented in Table 5.5. We contacted the dataset authors [164, 165] about this deficiency, unfortunately they are uncertain whether it was introduced during the construction of the dataset or during the preparation of the public version. Thus, it remains unknown if the fine-grained dataset authors were using bug reports with or without the missing descriptions in their experiments. To overcome this problem we decided to evaluate our method on both versions of the fine-grained dataset. For further analysis of missing descriptions impact see Section 5.7.

5.6.3 *Results for fine grained dataset*

We utilize the same data split into training and testing subsets as Ye et al. [165], using equal sized folds of 500 bug reports, where consecutive folds are used for training and evaluation. This gives us two folds for AspectJ and Tomcat, eight folds for BIRT and SWT and twelve folds for Eclipse UI and JDT.

All the results are presented in Table 5.6 and on Fig. 5.3. Compared to results reported in Ye et al. [165], for the pointwise learning to rank method we obtain better results in terms of Accuracy@1, MAP (Equation (6)) and MRR (Equation (7)). Results for the rest of Accuracy@k (Equation (5)) are comparable (i.e. slightly worse or better). We significantly improve results on AspectJ and BIRT projects. Adding missing descriptions yields better results in terms of Accuracy@k and MAP for AspectJ, Eclipse UI and Tomcat, while for other projects results are lower by about 1% on Accuracy@k. We outperform baseline methods BugLocator, VSM and UsualSuspect as reported in [165], both with and without missing description.

5.6.4 *BugLocator dataset*

This dataset [172] is commonly used by variety of the state-of-the-art methods, that can be found as part of Bug Center project[3]. There are four open source projects in this dataset: AspectJ, Eclipse 3.1, SWT 3.1, and ZXing. We evaluated our method on part of this dataset

2 http://dx.doi.org/10.6084/m9.figshare.951967
3 https://code.google.com/archive/p/bugcenter/

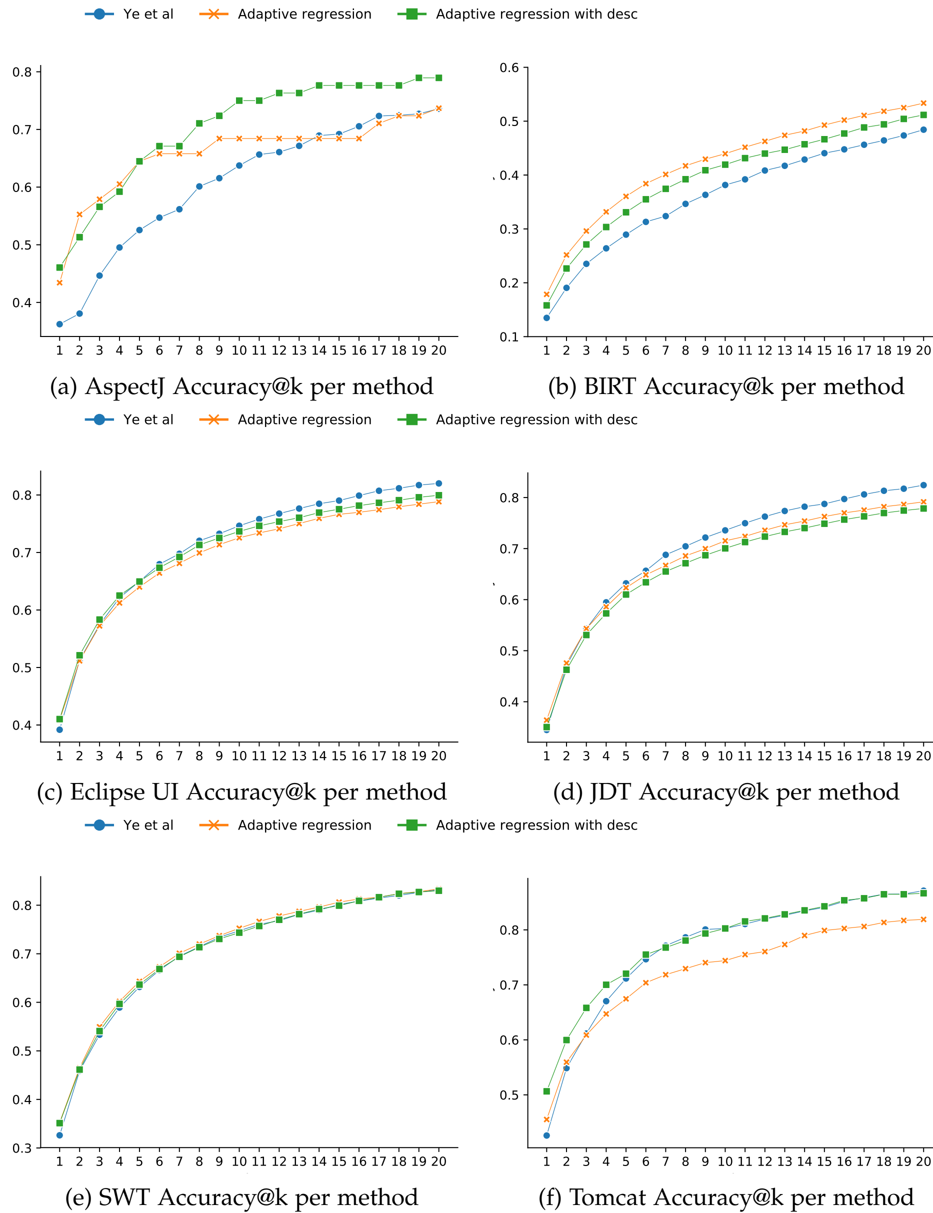


Figure 5.3: Accuracy@k (see Equation (5)) evaluation on fine-grained Ye et al. dataset [165] including results as reported in source publication [165]. Our method is able to outperform Ye et al. on Accuracy@1 on all projects. Results for Ye et al. were acquired by digitalizing Accuracy@k diagrams from [165] (as authors do not provide exact results). We omitted other commonly evaluated methods, BugLocator, VSM and UsualSuspect as they are outperformed by presented methods (see [165]).

Table 5.6: MAP (see Equation (6)) and MRR (see Equation (7)) evaluation on fine-grained Ye et al. dataset [165], including results reported therein. Our method is able to outperform Ye et al. on all projects. We do not present BugLocator, VSM and Usual Suspects as both Ye et al. outperform these methods. The best and second best results are underlined with solid line and dotted line respectively. We denote the version "with desc" on the dataset with restored missing bug report descriptions.

	MAP						MRR					
Method	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat	AspectJ	BIRT	Eclipse	JDT	SWT	Tomcat
Adaptive regression	0.45	0.21	0.44	0.40	0.42	0.50	0.53	0.27	0.52	0.48	0.48	0.56
Adaptive regression with desc	0.46	0.19	0.45	0.39	0.41	0.54	0.54	0.25	0.52	0.47	0.48	0.61
Ye et al.+ [165]	0.37	0.16	0.44	0.39	0.40	0.49	0.44	0.21	0.51	0.47	0.46	0.55

for completeness, as it has several disadvantages compared to the fine grained dataset. BugLocator dataset is based on a single version of project sources connected to multiple fixed bug reports, and contains fewer bug reports than the fine grained dataset. The authors did not include feature values, which necessities additional preprocessing. The repository history is not present, nor explicit bug fix connection using commit SHA identifier, only the fix commit date. In the case of the Eclipse 3.1 project the single version of source code is not based on repository, but on a prepared development package which includes partial source code. Furthermore, bug reports of this project are related to multiple existing Eclipse repositories such as Eclipse Platform UI or Eclipse JDT UI. Note, that this leads to misaligned file paths between bug reports and related repositories, hence it might affect performance of algorithms trained on the dataset. Moreover, Ye et al. [165] examined the BugLocator dataset and found files included in bug reports, but not present in single version of source code, as those files were deleted between fixing commit and single version preparation.

Similarly to Ye et al. [165], we use Eclipse 3.1, as other projects are too small to split them into folds. Then, as in [165], we split the data into six consecutive folds with 500 bug reports in each. We use fold n to train the $n+1$ fold, except for the first fold, for which the second fold is used for training.

5.6.5 *Results for BugLocator dataset*

For completeness we include results reported by other projects on this dataset, such as BugLocator [172], BRTTracer [158], BLUiR [128], Ye et al.+ [165], AmaLgam+ [155], ConCodeSe [28] and Shi et al. [131] [4]. It should be noted that Shi et al. [131] replicated results of BLUiR [128] and AmaLgam+ [155] on the BugLocator dataset, with lower results than original authors, nevertheless we cite original findings. We obtained compelling results, and outperformed other approaches, with the exception of Accuracy@10 for Shi et al. [131]. Results are gathered in Table 5.7, with highest results highlighted. The results were taken directly from the corresponding publication. Due to the fact that this dataset contains a single version, we did not use the past history of the repository when constructing ϕ_2.

4 For papers that reported top-n the conversion was made to Accuracy@k. For Ye et al. [165] we only report results from their extended work as they improve initial results.

Some experiments differ in experiment setup, thus are not directly comparable [172] [131, 158] (see Table 5.7), we include them for completeness.

Table 5.7: Method evaluation on Eclipse 3.1 project from single version BugLocator dataset [172], including results of other methods using the same dataset as in corresponding source publications. The best and second best results are highlighted in grey and light grey respectively. Setup differences: † training of randomly selected 5% reports, ‡ manually adjusted weights, ‡‡ usage of 30 folds, 100 bug reports each.

Method	Acc@1	Acc@5	Acc@10	MAP	MRR
BugLocator [172]	0.291	0.538	0.626	0.3	0.41
BRTTracer [158]	0.326	0.559	0.652	0.33	0.43
BLUiR [128]	0.329	0.562	0.654	0.33	0.44
Ye et al.+ [165]	0.34	0.57	0.66	0.34	0.45
AmaLgam+ [155]†	0.357	0.603	0.691	0.36	0.47
ConCodeSe [28]‡	0.376	0.612	0.699	0.37	0.57
Shi et al. [131]‡‡	0.297	0.664	0.85	0.306	0.399
Adaptive regression	0.7	0.752	0.785	0.6	0.728

5.6.6 *Performance*

We have conducted all experiments on two computers with different hardware setup.

- *Evaluation Setup:* 2 Ten-Core Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz processors, 62 GB RAM.
- *Dataset Setup:* 2 Six-Core AMD Opteron TM processors 2431 @ 2.40GHz, 32 GB RAM.

The results are gathered in Table 5.8. During experiments there was no other load on both machines.

On the *Evaluation Setup* we measured training time for our methods including all steps described in Section 4.4. The provided times include additional steps for parameter adaptation. Training times for proposed methods are below the main competitor method [165], but were achieved using more recent hardware. The feature computation time was measured on the *Dataset Setup* and is comparable to Ye et al. [165] in terms of average time per bug report. The *Dataset Setup* is similar to hardware used by Ye et al. [165]. Note that the feature computation time for both our method and Ye et al. [165] is the most time consuming step, and respective times are in our opinion acceptable for the intended use.

5.7 DISCUSSION

In this section we provide a detailed explanation of our findings. In particular we investigate the used learning to rank approach, explain how construction of training set affects the results, illustrate the parameter adaptation process and show threats to validity.

Table 5.8: Times are reported for fine-grained Ye et al. dataset [165]. Measurements are done using Python's `timeit` module. **Training:** Average time in seconds per bug report and per file, using 10 runs on each project, standard deviation lower than 0.01; done on hardware *Evaluation Setup*. **Feature computation:** Summary feature computation time per bug reported; done on hardware *Dataset Setup*.Results <(± 0.01)

Time		per	AspectJ	BIRT	Eclipse UI	JDT	SWT	Tomcat
Training	Regr.	bug rep.	0.321	0.338	0.345	0.353	0.328	0.330
		file	0.018	0.017	0.019	0.018	0.031	0.020
	Presc.	bug rep.	0.079	0.080	0.081	0.082	0.080	0.080
		file	0.004	0.004	0.004	0.004	0.008	0.005
Feat comp.		bug rep.	39.16	74.69	42.61	68.03	46.05	23.64

5.7.1 *Learn to rank approaches and training target construction*

The usefulness of pointwise, pairwise and listwise approaches in bug localization depends heavily on the training target function. Ye et al. [165] used relevant and irrelevant files as a pairwise training target for SVMrank. Such formulation is sufficient for pairwise and listwise approaches but does not fit well the pointwise approach. Shi et al. [131] evaluated all available learning to rank algorithms from RankLib [27]. The authors concluded that *Coordinate Ascent* being a listwise approach, which optimizes the MAP metric is the best suited method. They also noted poor performance of algorithms based on the pointwise principle, due to the formulation of the training target function. The AmaLgam+ [155] uses a genetic algorithm from the JGAP [98] library to find feature weights. Due to the fact that $e^{MAP+MRR}$ is optimized on all results, it is a listwise approach.

In our method we propose a custom training target function that better suits pointwise approaches. This target function is based on a score that assigns a value from the continuous range $[0, 1]$, which we then ameliorate to ensure that relevant files are on top of the list. Thus all relevant files have higher scores than irrelevant files, and the training target is in linear order. The reformulation of the training function allowed us to successfully apply pointwise learning to rank approach.

5.7.2 *Imbalance handling*

Both the ratio between defect and non-defect (irrelevant) files, and selection of which files are included in the training set affects learning performance. In particular, the training set needs to contain all defect files, and some subset of non-defect ones, to be able to generalize properly during the training process. Ye et al. [165] choose the training set irrelevant files *similar* to the bug report. The files are selected with the highest value of cosine similarity of file content and text of the report. They have found that using up to around 200 irrelevant files per bug report improves *MAP* metric.

The authors of BLUiR [128] use whole AspectJ project from BugLocator dataset [172] as a training set to establish weights for their method. Shi et al. [131] selected irrelevant files for training utilizing BLUiR [128]. The ConCodeSe [28] uses *randomly* selected 2.2% of bug

reports as a training set to adjust text similarity weights. The Authors of AmaLgam+ [155] method *randomly* sampled the whole dataset to find 5% bug reports for training.

Our model selects a portion of *dissimilar* irrelevant files. In particular we apply cutoff function to maximize training results (with adaptive cutoff), under the constraint that defect files make up the majority of the training set.

5.7.3 *Adaptation process*

Ye et al. [165] conducted manual grid search on one training fold to find optimal parameters for evaluation. Shi et al. [131] used default parameters present in RankLib [27] for all tested algorithms, with an additional manual grid search conducted on the first fold for Eclipse 3.1. The authors adjusted restarts and iterations parameters of Coordinate Ascent algorithm, choosing 5 and 25 to be used for the rest of the project. The AmaLgam+ [155] computes weights using genetic algorithm optimizing $e^{MAP+MRR}$ on train data, using default parameters of JGAP library [98]. Due to the fact that the characteristics of a project may change over time, models with predefined parameters may have suboptimal results or degrade during industrial usage. Our method is able to efficiently adapt the parameters over time from a predefined set of parameters. The used scoring functions and regression parameters are selected on the training fold (fold n) and then used on the testing fold (fold $n+1$).

Fold size is set the same way as in Ye et. al. [165] (500 bug reports) for comparison reasons. We investigated how different fold size can affect adaptation and training. In those additional experiments we used fold sizes of 100 and 250 bug reports on 4 large projects (Birt, Eclipse UI, JDT and SWT) from fine-grained dataset [165], and achieved comparable results of our method in terms of all metrics as with default size.

5.7.4 *Selected parameters*

Depending on the dataset different parameters are selected. For the fine grained dataset, our algorithm selects Levene test for scoring function (see Fig. 5.4) and Huber loss function [113] with cutoff factor of 5% (see Fig. 5.6). The dominant regularization function was L1 norm as seen on Fig. 5.5. The best cutoff factor for irrelevant files is 5% when training on the same data. For single version dataset our algorithm selects chi-square test for one fold and Levene test for remaining folds as scoring function. For corresponding regression model training the algorithm chooses Huber loss function and cutoff factor of 5%.

5.7.5 *Impact of feature construction differences*

To examine this impact we evaluated the replication code for [165] on different variants of features. These experiments were confined to AspectJ, SWT and Tomcat. The proposed feature ϕ_2^* does not leak future data which might cause lower results. We compared ϕ_2^* and ϕ_2, using the same replication code. For SWT and Tomcat, using ϕ_2^* resulted with lower results on all metrics (on average 6% and 7%). For AspectJ the results are higher: 1% for Accuracy@k on average and 3% on MAP, MRR difference below 1%. We did a similar experiment to assess impact of TF-IDF weighting schemes and tokenization. Our approach

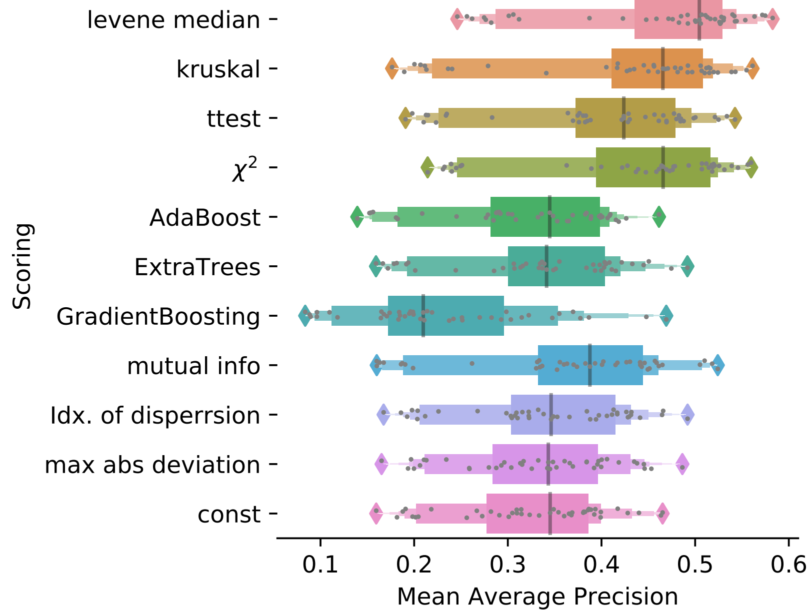


Figure 5.4: Mean Average Precision distribution on all training folds for all projects for different scoring functions. Distribution presented with letter-value plots [52], actual measurements shown as dots.

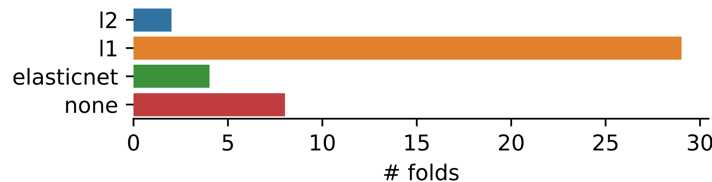


Figure 5.5: Adaptation of regression model regularization across all training folds on fine-grained Ye et al. dataset [165]

decreased replication results by 1%-2% for Tomcat and SWT projects across all metrics and improved them by 8% on AspectJ.

5.7.6 *Adaptive scoring*

Note that the initial ranking can be used as a standalone ranking algorithm. It is able to outperform Ye et al. [165] on AspectJ, Birt, JDT and SWT, with MAP of 0.45, 0.21, 0.40 and 0.41 respectively on Ye et al. [165] dataset with missing descriptions. For Eclipse Platform UI and Tomcat it obtains MAP of 0.43 and 0.48, slightly below [165]. Similarly the adaptive scoring is able to outperform other state–of–the–art approaches on BugLocator dataset [172], with MAP of 0.58 on Eclipse 3.1, slightly below the adaptive regression method.

5.7.7 *Threats to validity*

Missing bug report descriptions. All projects in the fine-grained dataset [165] are missing some of the bug report descriptions (see Section 5.6.2) that were present in Bugzilla. We have investigated the impact of missing descriptions by preparing a replication of Ye et

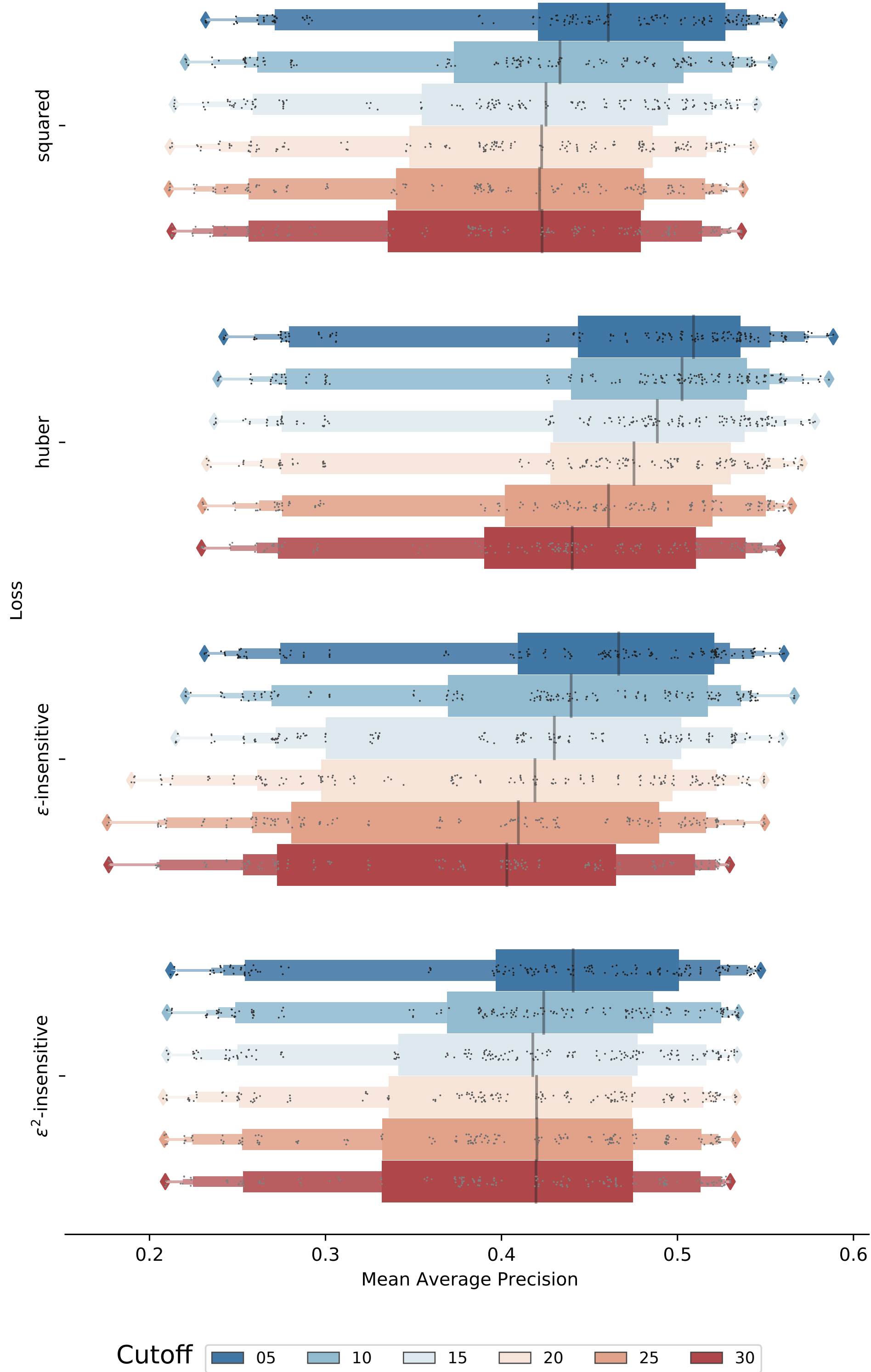


Figure 5.6: Distribution of Mean Average Precision for Adaptive Regression model. Evaluated on all training folds for each combination of loss function (Y-axis) and cutoff factor as % (cutoff's factors are grouped by loss, colours in groups are the in the same order as in legend). Distribution presented with letter-value plots [52], actual measurements shown as dots.

al. [165] based on the source code the authors sent us. We evaluated replication results

on AspectJ and Tomcat, and obtained results lower by 8%-12% for the dataset variant with missing descriptions. However using all descriptions produces similar results as Ye et al. [165] within a 1-4% absolute error margin, on all metrics.

Projects homogeneity. Most of the research in this topic (see Table 5.1) was conducted using two datasets [120, 165], which are based on mature open source Java projects. The majority of analyzed projects are maintained by the Eclipse Foundation and use the same instance of Bugzilla bug tracker, and similar practices of code review. The question is how this impacts existing methods. Therefore, this is a valid argument that a more diverse dataset is needed. Such a dataset should include projects created in different languages and using varying development practices.

Potential biases in dataset. We investigated the fine-grained dataset [165] for biases, such as bug reports which are wrongly classified or are already localized (ie. filename present in summary or description) [72, 100]. First, we found individual cases of bug reports pointing to the wrong project due to an incorrect bug id in the commit message. Excluding them does not have impact on the overall results. Secondly, Kochar et. al. [72] found that in some projects even 50% of bug reports may be already localized. This is not the case for used fine-grained dataset [165], as around 70% of bug reports include at least one filename, but only 25% of them may be localized that way.

Retrospective study. Without prospective evaluation, involving new bug reports and real developers it is hard to judge the usefulness of the proposed method. Unfortunately, prospective studies are hard to conduct in real life, and it may be difficult to compare results. A prospective study was presented in [154], where 58 students evaluated bug localization on fixed bugs in SWT. The data was gathered by surveying participants in the DebugRecorder IDE plugin. In contrast, all presented papers in Section 5.2 focus on retrospective analysis of bug localization. This allows comparable and replicable results. Thus our evaluation is done the same way.

5.8 CONCLUDING REMARKS

The bug localization system needs to recommend several likely files among all within the software project [62]. In our opinion, the learning–to–rank approach is a good solution to this ranking problem [131, 155, 164, 165], although different algorithms can also be used, for instance classification models [139]. In this study we propose a generalized view on a building block for such methods in the context of bug localization. Then, we propose a new adaptive method based on the pointwise principle.

We propose a new initial ranking scheme that facilitates the construction of a more robust training target function, which eventually allows successfully applying the pointwise learning to rank algorithm. Our method is designed in such a way that it will adapt its parameters to the characteristics of a project, without the need for separate parameters fitting procedures. Furthermore, it will adapt over time to changes in project characteristics.

In our experiments on two datasets we had shown that adaptive method is competitive with state-of-the-art [165]. In particular, we improve *Accuracy@1*, *MAP* and *MRR* metrics on all evaluated projects.

6 CONCLUSIONS

In this dissertation we examined several software engineering approaches based on Mining of Software Repositories in order to improve project quality. Our preliminary research on open source contributions and bug detection, shown in Chapter 3, inspired us to focus on code review and bug localization.

In particular, in Chapter 4 we have presented a novel profile based code reviewer recommendation method. We have shown improvements in both accuracy and performance over state-of-the-art, proposed by Thongtanunam et al. [145]. Consequently, the proposed method can be applied to large repositories of open source and industrial systems. Additionally, the presented approach is extensible, enabling the possibility of further improvements.

As presented in Chapter 5 we proposed adaptive bug localization approach using bug reports, which select best method without any manual tuning of the algorithm. We demonstrated the significance of our findings on two datasets [164, 172] commonly used in this area of research. Consequently, we obtained better results than the current state-of-the-art Ye et al. [165] method.

Finally, to support all our findings we use either publicly available datasources or we release computed features as replication data. Furthermore, we published source code of all our methods to simplify reproduction of our results.

6.1 FUTURE WORK

There are several possible future areas of research related to each research topic discussed in this dissertation. Moreover, recent advances in machine learning inspired us to seek applications of multi view approach in context on MSR.

In case of Chapter 3 we mainly see area to improve in dataset construction for bug detection method. By introducing alternatives to the *SZZ* algorithm, such as the tagging approach proposed by Treude et al. [147], different commits can be marked as defective further improving training quality. Another possible alternative to *SZZ* was proposed by Tian et al. [146] utilizing an ensemble of supervised SVM and semi-supervised "Learning from Positive and Unlabeled Examples" approach to find fixing commits.

For recommendation of code reviewers introduced in Chapter 4 we can propose the following improvements. Firstly, the currently used Thongtanunam dataset [145] can be further extended by adding the whole project repository. The source code and commit metadata from such repositories would enable creation of richer profiles. Secondly, profile data structure can be optimized, for example using locality-sensitive hashing can yield additional performance improvements. Thirdly, our results still are worth improving, for instance by using the results of other methods as ensemble. In our opinion collaborative filtering may lead to improvements of prediction accuracy. We did not use it in the current method due to the small number of developers compared to the quantity of file paths in all commits. Recent Yu et al. papers [167, 168] show that fusion and ensemble methods can achieve similar performance to the state-of-the-art. By introducing the social factors to our

method as a reviewer ordering mechanism we should be able to confirm or disprove this fact.

Concerning Chapter 5 we plan to expand adaptation possibilities of our bug localization approaches. Intuitively our method is extensible for additional scoring functions and regression models in respective steps. Additionally, the dataset can be further improved by preparation of new features and the addition of new projects. Such features can be based on cyclomatic complexity metrics, stack traces present in bug reports or natural language processing techniques such as n-grams obtained from both source code and report text. In case of evaluated projects, we would like to evaluate those written in different languages than Java.

As a possible direction of future research we recognize applying multi-view learning to MSR purposes. For every considered problem we used data from a single source, which corresponds to a single view on software repository and related artifacts. While such an approach was sufficient for our purposes, the underlying multiple sources of information can be treated utilizing multi-view machine learning in a semi-supervised manner. In such a case, each view can be handled by a separate algorithm, with correct results encountered by one algorithm being passed to others, reinforcing advance to optimal solution [140, 162].

7 APPENDIX: REPLICATION AND DATASETS

7.1 REPLICATION REPOSITORY

The required source code to replicate the findings presented in this dissertation can be downloaded from http://www-users.mat.umk.pl/~mfejzer/phd_replication. Alternatively, please use https://github.com/mfejzer/phd_replication or contact me directly using the following email address: mfejzer@mat.umk.pl.

The repository contains four directories:

- `contribution_analysis` - replication of experiments from Section 3.3 based on [35]
- `review_bug_detection` - replication of experiments from Section 3.4 based on [36]
- `reviewer_recommendation` - replication of experiments from Chapter 4 based on [34]
- `tracking_buggy_files` - replication of experiments from Chapter 5 based on [33]

In each directory there is a `README.md` file which describes the project, required dependencies, compilation process and how to conduct the experiments. Please consult the `LICENSE.txt` file in each directory for licensing details for source code and provided datasets.

7.2 PRELIMINARY STUDIES DATASETS

In this section we present datasets for Chapter 3:

- 42 selected projects of GHTorrent dataset [41] used for contribution analysis,
- 63 GitHub repositories and migrated Subversion repository utilized for bug detection.

Table 7.1: Contributor groups according to comment aggregation by author

Project	Commenters					Commiters
	Specialists	Generalists	Both	Other	Total	
bitcoin	2	5	2	9	14	133
boto	1	32	1	0	32	396
cakephp	1	9	1	16	25	291
CodeIgniter	1	5	1	22	27	308
compass	3	3	3	13	16	196
CraftBukkit	3	11	3	38	49	289
d3	1	1	1	1	2	58
devise	1	3	1	20	23	345
diaspora	4	19	4	47	66	401
django	2	18	2	23	41	417
django-cms	3	8	3	12	20	261
elasticsearch	1	2	1	10	12	124
facebook-android-sdk	3	10	3	0	10	39
foundation	3	4	3	33	37	359
gitlabhq	1	11	1	29	40	481
homebrew	2	17	2	92	109	3192
html5-boilerplate	3	5	3	29	34	267
jekyll	3	3	2	13	17	195
jquery	2	15	2	50	65	316
libgit2	2	7	2	9	16	147
libuv	1	3	1	6	9	107
MaNGOS	1	53	1	99	152	567
mongo	2	12	2	16	28	226
netty	2	2	2	10	12	117
node	1	16	1	48	64	590
openFrameworks	1	3	1	10	13	134
paperclip	2	4	2	26	30	255
phantomjs	2	4	2	10	14	115
php-sdk	1	4	1	4	8	38
phpunit	1	3	1	5	8	95
rails	3	8	3	12	20	2060
redis	1	1	1	4	5	120
requests	1	4	1	26	30	333
sbt	1	2	1	3	5	70
scala	3	9	2	22	32	142
Sick-Beard	1	3	1	21	24	183
SignalR	4	3	2	5	10	52
symfony	1	10	1	36	46	74
xphere-forks/symfony	1	4	1	26	30	731
three.js	1	4	1	21	25	258
tornado	1	1	1	9	10	149
TrinityCore	1	49	1	148	197	485
xbmc	3	26	3	58	84	390

Table 7.2: Bug prevalence according to topic aggregation by comment or issue month

Project	Commit comments				Issues			
	Months		Number		Months		Number	
	All	Bug related	All	Bug related	All	Bug related	All	Bug related
bitcoin	22	9	32	10	35	35	846	825
boto	29	11	25	12	39	39	545	508
cakephp	35	27	116	55	38	38	563	543
CodeIgniter	25	17	82	38	27	27	446	391
compass	31	13	38	20	53	53	721	705
CraftBukkit	32	32	336	219	34	34	470	402
d3	25	6	14	6	38	38	710	658
devise	34	22	62	33	44	44	1039	1015
diaspora	35	33	388	231	38	38	825	813
django	26	20	102	56	19	19	411	339
django-cms	31	12	38	16	55	55	891	890
elasticsearch	28	9	34	11	45	45	1066	1055
facebook-android-sdk	13	13	13	13	18	11	18	11
foundation	16	13	72	29	24	24	795	795
gitlabhq	24	20	135	56	25	25	845	845
homebrew	34	34	308	232	52	52	1694	1694
html5-boilerplate	33	27	139	86	44	44	756	754
jekyll	20	5	27	6	55	55	760	738
jquery	46	46	362	252	38	38	614	588
libgit2	35	19	73	32	37	37	652	639
libuv	28	24	93	48	31	31	455	423
MaNGOS	53	52	624	449	15	7	19	8
mongo	34	13	38	17	36	35	196	144
netty	24	22	163	68	36	36	569	563
node	50	42	321	173	39	39	729	700
openFrameworks	28	21	87	43	48	48	893	893
paperclip	40	31	76	41	54	53	679	612
phantomjs	16	6	17	6	34	34	563	533
php-sdk	12	6	11	6	21	4	18	6
phpunit	24	7	16	8	40	40	586	545
rails	31	11	32	16	55	55	893	892
redis	37	17	41	20	40	40	511	471
requests	25	11	52	18	33	33	652	613
sbt	22	1	3	1	34	34	444	435
scala	22	14	101	35	23	23	537	526
Sick-Beard	34	22	71	32	38	38	286	247
SignalR	19	6	28	8	28	28	664	655
symfony	41	40	254	162	38	38	1358	1358
three.js	36	32	237	132	42	42	1146	1146
tornado	26	11	25	12	49	49	537	489
TrinityCore	35	35	1122	1023	34	34	470	413
xbmc	38	38	476	379	34	34	808	803

Table 7.3: Small GitHub repositories used for bug detection

Name	History length	Revision	F1 for history limit					
			500		1000		1500	
			buggy	non buggy	buggy	non buggy	buggy	non buggy
flockdb	740	d0e72e8	0.681	0.902	0.545	0.844	0.545	0.844
beanstalkd	816	0fcf38b	0.394	0.667	0.657	0.726	0.657	0.726
octopress	857	5717a50	0.182	0.769	0.459	0.802	0.459	0.802
httpie	1112	fc497da	0.4	0.826	0.602	0.823	0.574	0.814
kestrel	1146	cc280ec	0.583	0.766	0.623	0.762	0.623	0.762
plupload	1181	03f6911	0.222	0.829	0.381	0.835	0.393	0.783
facebook-android-sdk	1234	f808e2ea	0.118	0.918	0.636	0.867	0.612	0.832
chosen	1282	0035be8	0.235	0.843	0.495	0.797	0.495	0.797
mosh	1368	b1da700	0.375	0.803	0.463	0.801	0.647	0.788
gizzard	1392	9b843254	0.667	0.934	0.447	0.851	0.467	0.845
ActionBarSherlock	1480	2c71339e	0.48	0.926	0.514	0.897	0.318	0.831
memcached	1566	1939cf9	0.727	0.786	0.68	0.69	0.717	0.688
RestSharp	1607	a8a9f34	0.647	0.633	0.704	0.523	0.752	0.484
html5-boilerplate	1740	ceb4620	0.453	0.803	0.511	0.734	0.432	0.667
paperclip	1970	6661480	0.279	0.803	0.432	0.742	0.619	0.795
ccv	2005	dbeaced7	0.364	0.687	0.5	0.68	0.537	0.71

Table 7.4: Medium GitHub repositories used for bug detection

Name	History length	Revision	F1 for history limit					
			500		1000		1500	
			buggy	non buggy	buggy	non buggy	buggy	non buggy
hiphop-php	3084	ae00e6e	0.514	0.897	0.556	0.731	0.653	0.686
clojure	3290	653b8465	0.261	0.904	0.35	0.928	0.321	0.894
scalatra	3353	54edddc7	0.414	0.901	0.427	0.836	0.432	0.814
compass	3536	4de01475	0.24	0.891	0.457	0.885	0.345	0.843
devise	3620	f48b6f16	0.121	0.826	0.255	0.881	0.168	0.844
flask	3762	a3f07829	0.407	0.752	0.623	0.764	0.61	0.769
Slim	3894	a24fac2f	0.609	0.794	0.611	0.781	0.612	0.755
d3	4195	171a607e	0	0.947	0.556	0.871	0.642	0.838
libuv	4423	ae12376d	0.581	0.812	0.578	0.785	0.567	0.725
shiny	4484	89bd7e90	0.368	0.852	0.383	0.751	0.563	0.783
SignalR	4673	61c1a688	0.557	0.806	0.531	0.739	0.576	0.711
SparkleShare	4678	cf446c00	0.44	0.813	0.649	0.86	0.503	0.775
knitr	5758	11ddfc6e	0.16	0.88	0.267	0.941	0.178	0.933
requests	5917	fab1fd10	0.571	0.968	0.333	0.922	0.319	0.874
egit	6111	576ac49ae	0.364	0.821	0.416	0.735	0.523	0.689
jquery	6421	1b74660f	0.702	0.605	0.758	0.504	0.776	0.516

Table 7.5: Large GitHub repositories used for bug detection

Name	History length	Revision	F1 for history limit					
			500		1000		1500	
			buggy	non buggy	buggy	non buggy	buggy	non buggy
boto	7198	91ba037e	0.4	0.951	0.204	0.889	0.323	0.866
sbt	7563	f72990123	0.564	0.894	0.522	0.9	0.525	0.906
folly	7644	b2955150	0.261	0.904	0.349	0.878	0.404	0.868
reddit	7956	753b17407	0.296	0.89	0.416	0.861	0.403	0.828
jEdit	8007	0cf98ba05	0.613	0.768	0.488	0.86	0.482	0.802
redis	8387	758b39be	0.489	0.852	0.404	0.83	0.551	0.827
netty	9523	d8b1a2d93f	0.294	0.855	0.464	0.792	0.461	0.722
CodeIgniter	10024	00df649ef	0.286	0.946	0.235	0.888	0.319	0.874
storm	10124	6afaa369	0.391	0.818	0.417	0.75	0.577	0.798
jekyll	10642	edc8f6b70	0.5	0.979	0.091	0.947	0.231	0.927
ServiceStack	10842	a3bd50229	0.444	0.913	0.269	0.891	0.263	0.893
phpunit	12521	4d2e3f801	0.194	0.852	0.421	0.904	0.336	0.856
libgit2	12685	0ec0b2bbd	0.32	0.903	0.361	0.833	0.493	0.837
django-cms	15808	3b5df9569	0.222	0.829	0.357	0.829	0.403	0.794
diaspora	19915	995f3394a	0.1	0.9	0.316	0.928	0.341	0.896
bitcoin	21529	a689c1190	0.4	0.951	0.308	0.925	0.175	0.873

Table 7.6: XL GitHub repositories and Subversion repository used for bug detection

Name	History length	Revision	F1 for history limit					
			500		1000		1500	
			buggy	non buggy	buggy	non buggy	buggy	non buggy
akka	24045	619f821e8d	0.465	0.854	0.396	0.822	0.455	0.79
zendframework	27058	8cc99f76cc	0.154	0.874	0.296	0.89	0.264	0.868
django	27387	103a6f4307	0.729	0.8	0.544	0.765	0.448	0.652
node	28235	344c5c454d	0.444	0.913	0.361	0.885	0.324	0.857
three.js	30122	04965b15d	0.5	0.944	0.314	0.9	0.442	0.907
httpd	31567	91fd63d927	0.25	0.935	0.375	0.915	0.418	0.896
scala	33843	238a16a058	0.2	0.958	0.375	0.915	0.349	0.891
cakephp	37201	448f30918f	0.294	0.855	0.192	0.82	0.378	0.805
symfony	44619	d7e5dd120b	0.417	0.92	0.302	0.893	0.409	0.892
elasticsearch	48388	72a59d41111	0.111	0.912	0.348	0.915	0.341	0.887
mongo	48496	ac796463d5	0.125	0.924	0.2	0.911	0.348	0.915
subversion	55865	837425	0.326	0.815	0.51	0.832	0.544	0.866
legacy-homebrew	63882	c092d647a	0	0.985	0.5	0.995	0	0.988
rails	74711	4ea7769044	0.286	0.946	0.065	0.921	0.239	0.904
gitlabhq	102122	de2ae315	0	0.964	0	0.916	0.19	0.905
mono	120396	d8040ba8652	0.25	0.813	0.462	0.841	0.403	0.828

BIBLIOGRAPHY

[1] Miltiadis Allamanis and Charles A. Sutton. "Mining source code repositories at massive scale using language modeling". In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta and Sunghun Kim. IEEE Computer Society, 2013, pp. 207–216. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624029.

[2] Giuliano Antoniol and Yann-Gaël Guéhéneuc. "Feature Identification: A Novel Approach and a Case Study". In: *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005, pp. 357–366. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.48.

[3] Giuliano Antoniol and Yann-Gaël Guéhéneuc. "Feature Identification: An Epidemiological Metaphor". In: *IEEE Transactions on Software Engineering* 32.9 (2006), pp. 627–641. DOI: 10.1109/TSE.2006.88.

[4] John Anvik, Lyndon Hiew and Gail C. Murphy. "Who should fix this bug?" In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Ed. by Leon J. Osterweil, H. Dieter Rombach and Mary Lou Soffa. ACM, 2006, pp. 361–370. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134336.

[5] Erik Arisholm, Lionel C. Briand and Eivind B. Johannessen. "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models". In: *Journal of Systems and Software* 83.1 (2010), pp. 2–17. DOI: 10.1016/j.jss.2009.06.055.

[6] Alberto Arteta, Nuria Gómez Blas and Luis Fernando de Mingo López. "Solving complex problems with a bioinspired model". In: *Engineering Applications of Artificial Intelligence* 24.6 (2011), pp. 919–927. DOI: 10.1016/j.engappai.2011.03.007.

[7] B. Ashok et al. "DebugAdvisor: a recommender system for debugging". In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by Hans van Vliet and Valérie Issarny. ACM, 2009, pp. 373–382. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595766.

[8] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar T. Devanbu and Abraham Bernstein. "The missing links: bugs and bug-fix commits". In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. 2010, pp. 97–106. DOI: 10.1145/1882291.1882308.

[9] Sushil Krishna Bajracharya, Joel Ossher and Cristina Videira Lopes. "Leveraging usage similarity for effective retrieval of examples in code repositories". In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. 2010, pp. 157–166. DOI: 10.1145/1882291.1882316.

[10] Vipin Balachandran. "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 931–940. ISBN: 978-1-4673-3076-3.

[11] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin and Daniel Tarlow. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[12] Andrew Begel, Yit Phang Khoo and Thomas Zimmermann. "Codebook: discovering and exploiting relationships in software repositories". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu and Sebastián Uchitel. ACM, 2010, pp. 125–134. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806821.

[13] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu and Sai Charan Koduru. "An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps". In: *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. Ed. by Anthony Cleve, Filippo Ricca and Maura Cerioli. IEEE Computer Society, 2013, pp. 133–143. ISBN: 978-1-4673-5833-0. DOI: 10.1109/CSMR.2013.23.

[14] Christian Bird et al. "Fair and balanced?: bias in bug-fix datasets". In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by Hans van Vliet and Valérie Issarny. ACM, 2009, pp. 121–130. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595716.

[15] Christian Bird et al. "The promises and perils of mining git". In: *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. Ed. by Michael W. Godfrey and Jim Whitehead. IEEE Computer Society, 2009, pp. 1–10. ISBN: 978-1-4244-3493-0. DOI: 10.1109/MSR.2009.5069475.

[16] David B. Bisant and James R. Lyle. "A Two-Person Inspection Method to Improve Programming Productivity". In: *IEEE Transactions on Software Engineering* 15.10 (1989), pp. 1294–1304. DOI: 10.1109/TSE.1989.559782.

[17] David M. Blei, Andrew Y. Ng and Michael I. Jordan. "Latent Dirichlet Allocation". In: *Journal of Machine Learning Research* 3 (2003), pp. 993–1022.

[18] Silvia Breu, Rahul Premraj, Jonathan Sillito and Thomas Zimmermann. "Information needs in bug reports: improving cooperation between developers and users". In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010*. Ed. by Kori Inkpen Quinn, Carl Gutwin and John C. Tang. ACM, 2010, pp. 301–310. ISBN: 978-1-60558-795-0. DOI: 10.1145/1718918.1718973.

[19] Cagatay Catal and Banu Diri. "A systematic review of software fault prediction studies". In: *Expert Systems with Applications* 36.4 (2009), pp. 7346–7354. DOI: 10.1016/j.eswa.2008.10.027.

[20] Laurent Charlin and Richard Zemel. "The Toronto paper matching system: an automated paper-reviewer assignment system". In: *ICML Workshop on Peer Reviewing and Publishing Models*. 2013.

[21] Kai Cheng, Limin Xiang, Mizuho Iwaihara, Haiyan Xu and Mukesh K. Mohania. "Time-Decaying Bloom Filters for Data Streams with Skewed Distributions". In: *15th International Workshop on Research Issues in Data Engineering (RIDE-SDMA 2005), Stream Data Mining and Applications, 3-7 April 2005, Tokyo, Japan*. IEEE Computer Society, 2005, pp. 63–69. ISBN: 0-7695-2390-0. DOI: 10.1109/RIDE.2005.15.

[22] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895.

[23] Don Conry, Yehuda Koren and Naren Ramakrishnan. "Recommender systems for the conference paper assignment problem". In: *Proceedings of the 2009 ACM Conference on Recommender Systems, RecSys 2009, New York, NY, USA, October 23-25, 2009*. Ed. by Lawrence D. Bergman, Alexander Tuzhilin, Robin D. Burke, Alexander Felfernig and Lars Schmidt-Thieme. ACM, 2009, pp. 357–360. ISBN: 978-1-60558-435-5. DOI: 10.1145/1639714.1639787.

[24] Cesar Couto et al. "BugMaps-Granger: a tool for visualizing and predicting bugs using Granger causality tests". In: *Journal of Software Engineering Research and Development* 2 (2014), p. 1. DOI: 10.1186/2195-1721-2-1.

[25] Marco D'Ambros, Michele Lanza and Romain Robbes. "An extensive comparison of bug prediction approaches". In: *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. Ed. by Jim Whitehead and Thomas Zimmermann. IEEE Computer Society, 2010, pp. 31–41. ISBN: 978-1-4244-6803-4. DOI: 10.1109/MSR.2010.5463279.

[26] Valentin Dallmeier and Thomas Zimmermann. "Extraction of Bug Localization Benchmarks from History". In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, pp. 433–436. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321702.

[27] Van Dang. *The Lemur Project - Wiki - RankLib*. https://sourceforge.net/p/lemur/wiki/RankLib/. The Lemur Project, [Online]. 2012. (Visited on 24/03/2020).

[28] Tezcan Dilshener, Michel Wermelinger and Yijun Yu. "Locating bugs without looking back". In: *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by Miryung Kim, Romain Robbes and Christian Bird. ACM, 2016, pp. 286–290. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901775.

[29] Janet Drake, Vahid Mashayekhi, John Riedl and Wei-Tek Tsai. "'A Distributed Collaborative Software Inspection Tool: Design, Prototype, and Early Trial". In: *Proceedings of the 30th Aerospace Sciences Conference*. 1991.

[30] Susan T. Dumais and Jakob Nielsen. "Automating the Assignment of Submitted Manuscripts to Reviewers". In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Copenhagen, Denmark, June 21-24, 1992*. Ed. by Nicholas J. Belkin, Peter Ingwersen and Annelise

Mark Pejtersen. ACM, 1992, pp. 233–244. ISBN: 0-89791-523-2. DOI: 10.1145/133160.133205.

[31] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron and Audris Mockus. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12. DOI: 10.1109/32.895984.

[32] Michael E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development". In: *IBM Systems Journal* 15.3 (1976), pp. 182–211. DOI: 10.1147/sj.153.0182.

[33] Mikolaj Fejzer, Jakub Narebski, Piotr Przymus and Krzysztof Stencel. "Tracking Buggy Files: New Efficient Adaptive Bug Localization Method". 2020. Manuscript.

[34] Mikolaj Fejzer, Piotr Przymus and Krzysztof Stencel. "Profile based recommendation of code reviewers". In: *Journal of Intelligent Information Systems* 50.3 (2018), pp. 597–619. DOI: 10.1007/s10844-017-0484-1.

[35] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski and Krzysztof Stencel. "Open Source Is a Continual Bugfixing by a Few". In: *Advances in Databases and Information Systems - 18th East European Conference, ADBIS 2014, Ohrid, Macedonia, September 7-10, 2014. Proceedings*. Ed. by Yannis Manolopoulos, Goce Trajcevski and Margita Kon-Popovska. Vol. 8716. Lecture Notes in Computer Science. Springer, 2014, pp. 153–162. ISBN: 978-3-319-10932-9. DOI: 10.1007/978-3-319-10933-6_12.

[36] Mikolaj Fejzer, Michal Wojtyna, Marta Burzanska, Piotr Wisniewski and Krzysztof Stencel. "Supporting Code Review by Automatic Detection of Potentially Buggy Changes". In: *Beyond Databases, Architectures and Structures - 11th International Conference, BDAS 2015, Ustroń, Poland, May 26-29, 2015, Proceedings*. Ed. by Stanislaw Kozielski, Dariusz Mrozek, Pawel Kasprowski, Bozena Malysiak-Mrozek and Daniel Kostrzewa. Vol. 521. Communications in Computer and Information Science. Springer, 2015, pp. 473–482. ISBN: 978-3-319-18421-0. DOI: 10.1007/978-3-319-18422-7_42.

[37] Jaroslav M. Fowkes et al. "Autofolding for Source Code Summarization". In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1095–1109. DOI: 10.1109/TSE.2017.2664836.

[38] Yoav Freund and Robert E. Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings*. Ed. by Paul M. B. Vitányi. Vol. 904. Lecture Notes in Computer Science. Springer, 1995, pp. 23–37. ISBN: 3-540-59119-2. DOI: 10.1007/3-540-59119-2_166.

[39] Simson Garfinkel. *History's worst software bugs*. https://www.wired.com/2005/11/historys-worst-software-bugs/. Accessed: 2019-10-26. 2005.

[40] Pierre Geurts, Damien Ernst and Louis Wehenkel. "Extremely randomized trees". In: *Machine Learning* 63.1 (2006), pp. 3–42. DOI: 10.1007/s10994-006-6226-1.

[41] Georgios Gousios. "The GHTorent dataset and tool suite". In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta and Sunghun Kim. IEEE Computer Society, 2013, pp. 233–236. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624034.

[42] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection". In: *Journal of Machine Learning Research* 3 (2003), pp. 1157–1182.

[43] Aric Hagberg, Pieter Swart and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[44] Mark A. Hall et al. "The WEKA data mining software: an update". In: *SIGKDD Explorations* 11.1 (2009). https://www.cs.waikato.ac.nz/ml/weka/, pp. 10–18. DOI: 10.1145/1656274.1656278.

[45] Kazuki Hamasaki et al. "Who does what during a code review? datasets of OSS peer review repositories". In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. Ed. by Thomas Zimmermann, Massimiliano Di Penta and Sunghun Kim. IEEE Computer Society, 2013, pp. 49–52. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624003.

[46] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel and Volker Gruhn. "Automatically recommending code reviewers based on their expertise: an empirical comparison". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel and Sarfraz Khurshid. ACM, 2016, pp. 99–110. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970306.

[47] Ahmed E Hassan. "The road ahead for mining software repositories". In: *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE. 2008, pp. 48–57.

[48] Ahmed E. Hassan and Tao Xie. "Software intelligence: the future of mining software engineering data". In: *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by Gruia-Catalin Roman and Kevin J. Sullivan. ACM, 2010, pp. 161–166. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882397.

[49] Trevor Hastie, Saharon Rosset, Ji Zhu and Hui Zou. "Multi-class adaboost". In: *Statistics and its Interface* 2.3 (2009), pp. 349–360.

[50] Trevor Hastie, Robert Tibshirani and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009. ISBN: 9780387848570.

[51] Abram Hindle, Neil A. Ernst, Michael W. Godfrey and John Mylopoulos. "Automated topic naming to support cross-project analysis of software maintenance activities". In: *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*. Ed. by Arie van Deursen, Tao Xie and Thomas Zimmermann. ACM, 2011, pp. 163–172. ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985466.

[52] Heike Hofmann, Hadley Wickham and Karen Kafadar. "Letter-Value Plots: Boxplots for Large Data". In: *Journal of Computational and Graphical Statistics* 26.3 (2017), pp. 469–477. DOI: 10.1080/10618600.2017.1305277.

[53] Thomas Hofmann. “Probabilistic Latent Semantic Indexing”. In: *SIGIR ’99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA*. Ed. by Fredric C. Gey, Marti A. Hearst and Richard M. Tong. ACM, 1999, pp. 50–57. ISBN: 1-58113-096-1. DOI: 10.1145/312624.312649.

[54] George Hripcsak and Adam S. Rothschild. “Technical Brief: Agreement, the F-Measure, and Reliability in Information Retrieval”. In: *Journal of the American Medical Informatics Association* 12.3 (2005), pp. 296–298. DOI: 10.1197/jamia.M1733.

[55] Xuan Huo, Ming Li and Zhi-Hua Zhou. “Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by Subbarao Kambhampati. IJCAI/AAAI Press, 2016, pp. 1606–1612. ISBN: 978-1-57735-770-4.

[56] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel and Foutse Khomh. “Mining the relationship between anti-patterns dependencies and fault-proneness”. In: *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. Ed. by Ralf Lämmel, Rocco Oliveto and Romain Robbes. IEEE Computer Society, 2013, pp. 351–360. ISBN: 978-1-4799-2931-3. DOI: 10.1109/WCRE.2013.6671310.

[57] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann and Kwangkeun Yi. “Improving code review by predicting reviewers and acceptance of patches”. In: *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006)* (2009), pp. 1–18.

[58] Yue Jia and Mark Harman. “Constructing Subtle Faults Using Higher Order Mutation Testing”. In: *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*. IEEE Computer Society, 2008, pp. 249–258. ISBN: 978-0-7695-3353-7. DOI: 10.1109/SCAM.2008.36.

[59] Thorsten Joachims. “Text Categorization with Support Vector Machines: Learning with Many Relevant Features”. In: *Machine Learning: ECML-98, 10th European Conference on Machine Learning, Chemnitz, Germany, April 21-23, 1998, Proceedings*. Ed. by Claire Nedellec and Céline Rouveirol. Vol. 1398. Lecture Notes in Computer Science. Springer, 1998, pp. 137–142. ISBN: 3-540-64417-2. DOI: 10.1007/BFb0026683.

[60] Thorsten Joachims. “Training linear SVMs in linear time”. In: *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*. Ed. by Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven and Dimitrios Gunopulos. ACM, 2006, pp. 217–226. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150429.

[61] James A. Jones and Mary Jean Harrold. “Empirical evaluation of the tarantula automatic fault-localization technique”. In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. Ed. by David F. Redmiles, Thomas Ellman and Andrea Zisman. ACM, 2005, pp. 273–282. ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101949.

[62] Huzefa H. Kagdi, Michael L. Collard and Jonathan I. Maletic. "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". In: *Journal of Software Maintenance* 19.2 (2007), pp. 77–131. DOI: 10.1002/smr.344.

[63] Huzefa H. Kagdi, Jonathan I. Maletic and Bonita Sharif. "Mining Software Repositories for Traceability Links". In: *15th International Conference on Program Comprehension (ICPC 2007), June 26-29, 2007, Banff, Alberta, Canada*. IEEE Computer Society, 2007, pp. 145–154. ISBN: 0-7695-2860-0. DOI: 10.1109/ICPC.2007.28.

[64] Eirini Kalliamvakou et al. "The promises and perils of mining GitHub". In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim and Martin Pinzger. ACM, 2014, pp. 92–101. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597074.

[65] Dongsun Kim, Jaechang Nam, Jaewoo Song and Sunghun Kim. "Automatic patch generation learned from human-written patches". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 802–811. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606626.

[66] Dongsun Kim, Yida Tao, Sunghun Kim and Andreas Zeller. "Where Should We Fix This Bug? A Two-Phase Recommendation Model". In: *IEEE Transactions on Software Engineering* 39.11 (2013), pp. 1597–1610. DOI: 10.1109/TSE.2013.24.

[67] Sunghun Kim, E. James Whitehead Jr. and Yi Zhang. "Classifying Software Changes: Clean or Buggy?" In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196. DOI: 10.1109/TSE.2007.70773.

[68] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr. and Andreas Zeller. "Predicting Faults from Cached History". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 489–498. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.66.

[69] Sunghun Kim, Thomas Zimmermann, Kai Pan and E. James Whitehead Jr. "Automatic Identification of Bug-Introducing Changes". In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 2006, pp. 81–90. ISBN: 0-7695-2579-2. DOI: 10.1109/ASE.2006.23.

[70] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN: 0-201-03822-6.

[71] Pavneet Singh Kochhar, Tien-Duy B. Le and David Lo. "It's not a bug, it's a feature: does misclassification affect bug localization?" In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim and Martin Pinzger. ACM, 2014, pp. 296–299. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597105.

[72] Pavneet Singh Kochhar, Yuan Tian and David Lo. "Potential biases in bug localization: do they matter?" In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, 2014*. Ed. by Ivica Crnkovic, Marsha Chechik and Paul Grünbacher. ACM, 2014. DOI: 10.1145/2642937.2642997.

[73] William H Kruskal and W Allen Wallis. "Use of ranks in one-criterion variance analysis". In: *Journal of the American statistical Association* 47.260 (1952), pp. 583–621.

[74] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen and Tien N. Nguyen. "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*. Ed. by Myra B. Cohen, Lars Grunske and Michael Whalen. IEEE Computer Society, 2015, pp. 476–481. ISBN: 978-1-5090-0025-8. DOI: 10.1109/ASE.2015.73.

[75] David Landsberg, Hana Chockler and Daniel Kroening. "Probabilistic Fault Localisation". In: *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*. Ed. by Roderick Bloem and Eli Arbel. Vol. 10028. Lecture Notes in Computer Science. 2016, pp. 65–81. ISBN: 978-3-319-49051-9. DOI: 10.1007/978-3-319-49052-6_5.

[76] Alina Lazar, Sarah Ritchey and Bonita Sharif. "Improving the accuracy of duplicate bug report detection using textual similarity measures". In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim and Martin Pinzger. ACM, 2014, pp. 308–311. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597088.

[77] Tien-Duy B. Le, David Lo, Claire Le Goues and Lars Grunske. "A learning-to-rank based fault localization approach using likely invariants". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016, pp. 177–188. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931049.

[78] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung and Yves Le Traon. "Bench4BL: reproducibility study on the performance of IR-based bug localization". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 61–72. DOI: 10.1145/3213846.3213856.

[79] John Boaz Lee, Akinori Ihara, Akito Monden and Ken-ichi Matsumoto. "Patch Reviewer Recommendation in OSS Projects". In: *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*. Ed. by Pornsiri Muenchaisri and Gregg Rothermel. IEEE Computer Society, 2013, pp. 1–6. ISBN: 978-1-4799-2143-0. DOI: 10.1109/APSEC.2013.103.

[80] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim and Hoh Peter In. "Micro interaction metrics for defect prediction". In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, 2011, pp. 311–321. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025156.

[81] Meir M. Lehman. "On understanding laws, evolution, and conservation in the large-program life cycle". In: *Journal of Systems and Software* 1 (1980), pp. 213–221. DOI: 10.1016/0164-1212(79)90022-0.

[82] Howard Levene. "Contributions to probability and statistics". In: *Essays in honor of Harold Hotelling* (1960), pp. 278–292.

[83] Howard Levene. "Robust tests for equality of variances1". In: *Contributions to probability and statistics: Essays in honor of Harold Hotelling* 2 (1960), pp. 278–292.

[84] Chris Lewis et al. "Does bug prediction support human developers? findings from a google case study". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 372–381. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606583.

[85] Zhixing Li, Gang Yin, Yue Yu, Tao Wang and Huaimin Wang. "Detecting Duplicate Pull-requests in GitHub". In: *Proceedings of the 9th Asia-Pacific Symposium on Internetware, Internetware 2017, Shanghai, China, September 23 - 23, 2017*. Ed. by Hong Mei, Jian Lyu, Zhi Jin and Wenyun Zhao. ACM, 2017, 20:1–20:6. ISBN: 978-1-4503-5313-7. DOI: 10.1145/3131704.3131725.

[86] Dekang Lin. "An Information-Theoretic Definition of Similarity". In: *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*. Ed. by Jude W. Shavlik. Morgan Kaufmann, 1998, pp. 296–304. ISBN: 1-55860-556-8.

[87] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han and Samuel P. Midkiff. "SOBER: statistical model-based bug localization". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 286–295. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081753.

[88] Tie-Yan Liu. "Learning to Rank for Information Retrieval". In: *Foundations and Trends in Information Retrieval* 3.3 (2009), pp. 225–331. DOI: 10.1561/1500000016.

[89] Tie-Yan Liu. *Learning to Rank for Information Retrieval*. Springer, 2011. ISBN: 978-3-642-14266-6. DOI: 10.1007/978-3-642-14267-3.

[90] Xiang Liu, Torsten Suel and Nasir D. Memon. "A robust model for paper reviewer assignment". In: *Eighth ACM Conference on Recommender Systems, RecSys '14, Foster City, Silicon Valley, CA, USA - October 06 - 10, 2014*. Ed. by Alfred Kobsa, Michelle X. Zhou, Martin Ester and Yehuda Koren. ACM, 2014, pp. 25–32. ISBN: 978-1-4503-2668-1. DOI: 10.1145/2645710.2645749.

[91] V. Benjamin Livshits and Thomas Zimmermann. "DynaMine: finding common error patterns by mining software revision histories". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 296–305. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081754.

[92] Edward Loper and Steven Bird. "NLTK: the natural language toolkit". In: *arXiv preprint cs/0205028* (2002). https://www.nltk.org/.

[93] Andrian Marcus, Andrey Sergeyev, Václav Rajlich and Jonathan I. Maletic. "An Information Retrieval Approach to Concept Location in Source Code". In: *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*. IEEE Computer Society, 2004, pp. 214–223. ISBN: 0-7695-2243-2. DOI: 10.1109/WCRE.2004.10.

[94] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* 2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[95] Andrew Kachites McCallum. "MALLET: A Machine Learning for Language Toolkit". http://mallet.cs.umass.edu. 2002.

[96] Shane McIntosh, Yasutaka Kamei, Bram Adams and Ahmed E. Hassan. "The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects". In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim and Martin Pinzger. ACM, 2014, pp. 192–201. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597076.

[97] Wes McKinney. "pandas: a foundational Python library for data analysis and statistics". In: *Python for High Performance and Scientific Computing* 14 (2011). https://pandas.pydata.org/.

[98] Klaus Meffert. *The JGAP library*. https://sourceforge.net/projects/jgap/. [Online]. 2015. (Visited on 24/03/2020).

[99] Tim Menzies, Jeremy Greenwald and Art Frank. "Data Mining Static Code Attributes to Learn Defect Predictors". In: *IEEE Transactions on Software Engineering* 33.1 (2007), pp. 2–13. DOI: 10.1109/TSE.2007.256941.

[100] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota and Sonia Haiduc. "Are Bug Reports Enough for Text Retrieval-Based Bug Localization?" In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, 2018*. IEEE Computer Society, 2018. DOI: 10.1109/ICSME.2018.00046.

[101] Audris Mockus, Roy T. Fielding and James D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla". In: *ACM Transactions on Software Engineering and Methodology* 11.3 (2002), pp. 309–346. DOI: 10.1145/567793.567795.

[102] Adriaan Moors, Frank Piessens and Martin Odersky. "Parser combinators in Scala". In: *CW Reports* (2008). https://lirias.kuleuven.be/retrieve/13262.

[103] Laura Moreno et al. "Query-based configuration of text retrieval solutions for software engineering tasks". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman and Patrick Heymans. ACM, 2015. DOI: 10.1145/2786805.2786859.

[104] Raimund Moser, Witold Pedrycz and Giancarlo Succi. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer and Volker Gruhn. ACM, 2008, pp. 181–190. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368114.

[105] Emerson R. Murphy-Hill, Thomas Zimmermann, Christian Bird and Nachiappan Nagappan. "The design of bug fixes". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng and Klaus Pohl. IEEE Computer Society, 2013, pp. 332–341. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606579.

[106] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig and Brendan Murphy. "Change Bursts as Defect Predictors". In: *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. IEEE Computer Society, 2010, pp. 309–318. ISBN: 978-0-7695-4255-3. DOI: 10.1109/ISSRE.2010.25.

[107] Maliha S. Nash. "Handbook of Parametric and Nonparametric Statistical Procedures". In: *Technometrics* 43.3 (2001), p. 374. DOI: 10.1198/tech.2001.s629.

[108] Stacy D. Nelson and Johann Schumann. "What Makes a Code Review Trustworthy?" In: *37th Hawaii International Conference on System Sciences (HICSS-37 2004), CD-ROM / Abstracts Proceedings, 5-8 January 2004, Big Island, HI, USA*. IEEE Computer Society, 2004. ISBN: 0-7695-2056-1. DOI: 10.1109/HICSS.2004.1265711.

[109] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen and Tien N. Nguyen. "A topic-based approach for narrowing the search space of buggy files from a bug report". In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. Ed. by Perry Alexander, Corina S. Pasareanu and John G. Hosking. IEEE Computer Society, 2011, pp. 263–272. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100062.

[110] Travis E Oliphant. *A guide to NumPy*. Vol. 1. https://numpy.org/. Trelgol Publishing USA, 2006.

[111] Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell. "Predicting the Location and Number of Faults in Large Software Systems". In: *IEEE Transactions on Software Engineering* 31.4 (2005), pp. 340–355. DOI: 10.1109/TSE.2005.49.

[112] Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell. "Programmer-based fault prediction". In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE 2010, Timisoara, Romania, September 12-13, 2010*. Ed. by Tim Menzies and Günes Koru. ACM, 2010, p. 19. ISBN: 978-1-4503-0404-7. DOI: 10.1145/1868328.1868357.

[113] Art B Owen. "A robust hybrid of lasso and ridge regression". In: *Contemporary Mathematics* 443.7 (2007), pp. 59–72.

[114] Matheus Paixão, Jens Krinke, DongGyun Han and Mark Harman. "CROP: linking code reviews to source code changes". In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei and Emily Hill. ACM, 2018, pp. 46–49. DOI: 10.1145/3196398.3196466.

[115] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia and Rocco Oliveto. "Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells". In: *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 2016, pp. 244–255. ISBN: 978-1-5090-3806-0. DOI: 10.1109/ICSME.2016.27.

[116] Kai Pan, Sunghun Kim and E. James Whitehead Jr. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14.3 (2009), pp. 286–315. DOI: 10.1007/s10664-008-9077-5.

[117] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011). https://scikit-learn.org/, pp. 2825–2830.

[118] Alexey B Petrovsky. "An axiomatic approach to metrization of multiset space". In: *Multiple Criteria Decision Making: Proceedings of the Tenth International Conference: Expand and Enrich the Domains of Thinking and Application*. Springer Science & Business Media. 2012, p. 129.

[119] Danijel Radjenovic, Marjan Hericko, Richard Torkar and Ales Zivkovic. "Software fault prediction metrics: A systematic literature review". In: *Information & Software Technology* 55.8 (2013), pp. 1397–1418. DOI: 10.1016/j.infsof.2013.02.009.

[120] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl T. Barr and Premkumar T. Devanbu. "BugCache for inspections: hit or miss?" In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, 2011, pp. 322–331. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025157.

[121] Baishakhi Ray, Daryl Posnett, Vladimir Filkov and Premkumar T. Devanbu. "A large scale study of programming languages and code quality in github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso and Margaret-Anne D. Storey. ACM, 2014, pp. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922.

[122] Peter C. Rigby and Christian Bird. "Convergent contemporary software peer review practices". In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi and Mira Mezini. ACM, 2013, pp. 202–212. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491444.

[123] Peter C. Rigby, Daniel M. Germán and Margaret-Anne D. Storey. "Open source software peer review practices: a case study of the apache server". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer and Volker Gruhn. ACM, 2008, pp. 541–550. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368162.

[124] Peter C. Rigby and Margaret-Anne D. Storey. "Understanding broadcast based peer review on open source software projects". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall and Nenad Medvidovic. ACM, 2011, pp. 541–550. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985867.

[125] Brian C Ross. "Mutual information between discrete and continuous data sets". In: *PloS one* 9.2 (2014), e87357. DOI: 10.1371/journal.pone.0087357.

[126] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010. ISBN: 978-0-13-207148-2.

[127] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko and Alberto Bacchelli. "Modern code review: a case study at google". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Frances Paulisch and Jan Bosch. ACM, 2018, pp. 181–190. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183525.

[128] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid and Dewayne E. Perry. "Improving bug localization using structured information retrieval". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. Ed. by Ewen Denney, Tevfik Bultan and Andreas Zeller. IEEE, 2013, pp. 345–355. DOI: 10.1109/ASE.2013.6693093.

[129] Simone Scalabrino et al. "Automatically assessing code understandability: how far are we?" In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. Ed. by Grigore Rosu, Massimiliano Di Penta and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 417–427. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115654.

[130] Michael J. Scialdone, Na Li, Robert Heckman and Kevin Crowston. "Group Maintenance Behaviors of Core and Peripherial Members of Free/Libre Open Source Software Teams". In: *Open Source Ecosystems: Diverse Communities Interacting, 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings*. Ed. by Cornelia Boldyreff, Kevin Crowston, Björn Lundell and Anthony I. Wasserman. Vol. 299. IFIP Advances in Information and Communication Technology. Springer, 2009, pp. 298–309. ISBN: 978-3-642-02031-5. DOI: 10.1007/978-3-642-02032-2_26.

[131] Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin and Xingjun Zhang. "Comparing learning to rank techniques in hybrid bug localization". In: *Applied Soft Computing* 62 (2018), pp. 636–648. DOI: 10.1016/j.asoc.2017.10.048.

[132] Yonghee Shin, Robert M. Bell, Thomas J. Ostrand and Elaine J. Weyuker. "Does calling structure information improve the accuracy of fault prediction?" In: *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*. Ed. by Michael W. Godfrey and Jim Whitehead. IEEE Computer Society, 2009, pp. 61–70. ISBN: 978-1-4244-3493-0. DOI: 10.1109/MSR.2009.5069481.

[133] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella and Sunghun Kim. "Reducing Features to Improve Code Change-Based Bug Prediction". In: *IEEE Transactions on Software Engineering* 39.4 (2013), pp. 552–569. DOI: 10.1109/TSE.2012.43.

[134] D Singh, A Ibrahim, T Yohanna and J Singh. "An overview of the applications of multisets". In: *Novi Sad Journal of Mathematics* 37.3 (2007), pp. 73–92.

[135] Amit Singhal. "Modern Information Retrieval: A Brief Overview". In: *IEEE Data Engineering Bulletin* 24.4 (2001). http://sites.computer.org/debull/A01DEC-CD.pdf, pp. 35–43.

[136] Jacek Sliwerski, Thomas Zimmermann and Andreas Zeller. "When do changes induce fixes?" In: *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005. ISBN: 1-59593-123-6. DOI: 10.1145/1083142.1083147.

[137] Alexander J. Smola and Bernhard Schölkopf. "A tutorial on support vector regression". In: *Statistics and Computing* 14.3 (2004), pp. 199–222. DOI: 10.1023/B:STCO.0000035301.49549.88.

[138] Ian Sommerville. *Software engineering, 8th Edition*. International computer science series. Addison-Wesley, 2007. ISBN: 9780321313799.

[139] Higor Amario de Souza, Marcos Lordello Chaim and Fabio Kon. "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges". In: *CoRR* abs/1607.04347 (2016). arXiv: 1607.04347.

[140] Shiliang Sun. "A survey of multi-view machine learning". In: *Neural Computing and Applications* 23.7-8 (2013), pp. 2031–2038. DOI: 10.1007/s00521-013-1362-6.

[141] Wenbin Tang, Jie Tang and Chenhao Tan. "Expertise Matching via Constraint-Based Optimization". In: *2010 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2010, Toronto, Canada, August 31 - September 3, 2010, Main Conference Proceedings*. Ed. by Jimmy Xiangji Huang, Irwin King, Vijay V. Raghavan and Stefan Rueger. IEEE Computer Society, 2010, pp. 34–41. ISBN: 978-0-7695-4191-4. DOI: 10.1109/WI-IAT.2010.133.

[142] Camillo J Taylor. "On the optimal assignment of conference papers to reviewers". In: University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-30 (2008). https://www.seas.upenn.edu/~cjtaylor/PUBLICATIONS/pdfs/TaylorTR08.pdf.

[143] Ricardo Terra et al. "Measuring the Structural Similarity between Source Code Entities (S)". In: *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013*. Knowledge Systems Institute Graduate School, 2013, pp. 753–758.

[144] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida and Hajimu Iida. "Improving code review effectiveness through reviewer recommendations". In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2014, Hyderabad, India, June 2-3, 2014*. Ed. by Helen Sharp, Rafael Prikladnicki, Andrew Begel and Cleidson R. B. de Souza. ACM, 2014, pp. 119–122. ISBN: 978-1-4503-2860-9. DOI: 10.1145/2593702.2593705.

[145] Patanamon Thongtanunam et al. "Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review". In: *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. Ed. by Yann-Gaël Guéhéneuc, Bram Adams and Alexander Serebrenik. IEEE, 2015, pp. 141–150. ISBN: 978-1-4799-8469-5. DOI: 10.1109/SANER.2015.7081824.

[146] Yuan Tian, Julia L. Lawall and David Lo. "Identifying Linux bug fixing patches". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy and Mauro Pezzè. IEEE Computer Society, 2012, pp. 386–396. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227176.

[147] Christoph Treude and Margaret-Anne D. Storey. "Work Item Tagging: Communicating Concerns in Collaborative Software Development". In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 19–34. DOI: 10.1109/TSE.2010.91.

[148] Asher Trockman et al. ""Automatically assessing code understandability" reanalyzed: combined metrics matter". In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei and Emily Hill. ACM, 2018, pp. 314–318. DOI: 10.1145/3196398.3196441.

[149] Olivier Vandecruys et al. "Mining software repositories for comprehensible software fault prediction models". In: *Journal of Systems and Software* 81.5 (2008), pp. 823–839. DOI: 10.1016/j.jss.2007.07.034.

[150] Mario Linares Vásquez et al. "Triaging incoming change requests: Bug or commit history, or code authorship?" In: *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 451–460. ISBN: 978-1-4673-2313-0. DOI: 10.1109/ICSM.2012.6405306.

[151] Erik van der Veen, Georgios Gousios and Andy Zaidman. "Automatically Prioritizing Pull Requests". In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. IEEE, 2015, pp. 357–361. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.40.

[152] Ellen M. Voorhees. "The TREC-8 Question Answering Track Report". In: *Proceedings of The Eighth Text REtrieval Conference, TREC 1999, Gaithersburg, Maryland, USA, November 17-19, 1999*. Ed. by Ellen M. Voorhees and Donna K. Harman. Vol. Special Publication 500-246. National Institute of Standards and Technology (NIST), 1999.

[153] Fan Wang, Ben Chen and Zhaowei Miao. "A Survey on Reviewer Assignment Problem". In: *New Frontiers in Applied Artificial Intelligence, 21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2008, Wroclaw, Poland, June 18-20, 2008, Proceedings*. Ed. by Ngoc Thanh Nguyen, Leszek Borzemski, Adam Grzech and Moonis Ali. Vol. 5027. Lecture Notes in Computer Science. Springer, 2008, pp. 718–727. ISBN: 978-3-540-69045-0. DOI: 10.1007/978-3-540-69052-8_75.

[154] Qianqian Wang, Chris Parnin and Alessandro Orso. "Evaluating the usefulness of IR-based fault localization techniques". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. Ed. by Michal Young and Tao Xie. ACM, 2015. DOI: 10.1145/2771783.2771797.

[155] Shaowei Wang and David Lo. "AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization". In: *Journal of Software: Evolution and Process* 28.10 (2016), pp. 921–942. DOI: 10.1002/smr.1801.

[156] Shaowei Wang and David Lo. "Version history, similar report, and structure: putting them together for improved bug localization". In: *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*. Ed. by Chanchal K. Roy, Andrew Begel and Leon Moonen. ACM, 2014, pp. 53–63. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597148.

[157] David M. Weiss and Victor R. Basili. "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory". In: *IEEE Transactions on Software Engineering* 11.2 (1985), pp. 157–168. DOI: 10.1109/TSE.1985.232190.

[158] Chu-Pan Wong et al. "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 181–190. ISBN: 978-0-7695-5303-0. DOI: 10.1109/ICSME.2014.40.

[159] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu and Bhavani M. Thuraisingham. "Effective Software Fault Localization Using an RBF Neural Network". In: *IEEE Transactions on Reliability* 61.1 (2012), pp. 149–169. DOI: 10.1109/TR.2011.2172031.

[160] Rongxin Wu, Hongyu Zhang, Sunghun Kim and Shing-Chi Cheung. "ReLink: recovering links between bugs and changes". In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. Ed. by Tibor Gyimóthy and Andreas Zeller. ACM, 2011, pp. 15–25. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025120.

[161] Hong Xie and John C. S. Lui. "Mathematical Modeling of Competitive Group Recommendation Systems with Application to Peer Review Systems". In: *CoRR* abs/1204.1832 (2012). arXiv: 1204.1832.

[162] Chang Xu, Dacheng Tao and Chao Xu. "A Survey on Multi-view Learning". In: *CoRR* abs/1304.5634 (2013). arXiv: 1304.5634.

[163] David Yarowsky and Radu Florian. "Taking the load off the conference chairs: towards a digital paper-routing assistant". In: *1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*. https://www.aclweb.org/anthology/W99-0627. Citeseer, 1999.

[164] Xin Ye, Razvan C. Bunescu and Chang Liu. "Learning to rank relevant files for bug reports using domain knowledge". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso and Margaret-Anne D. Storey. ACM, 2014, pp. 689–699. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635874.

[165] Xin Ye, Razvan C. Bunescu and Chang Liu. "Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation". In: *IEEE Transactions on Software Engineering* 42.4 (2016), pp. 379–402. DOI: 10.1109/TSE.2015.2479232.

[166] Klaus Changsun Youm, June Ahn and Eunseok Lee. “Improved bug localization based on code change histories and bug reports”. In: *Information & Software Technology* 82 (2017), pp. 177–192. DOI: 10.1016/j.infsof.2016.11.002.

[167] Yue Yu, Huaimin Wang, Gang Yin and Charles X. Ling. “Reviewer Recommender of Pull-Requests in GitHub”. In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 609–612. ISBN: 978-0-7695-5303-0. DOI: 10.1109/ICSME.2014.107.

[168] Yue Yu, Huaimin Wang, Gang Yin and Tao Wang. “Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?” In: *Information & Software Technology* 74 (2016), pp. 204–218. DOI: 10.1016/j.infsof.2016.01.004.

[169] Feng Zhang, Audris Mockus, Iman Keivanloo and Ying Zou. “Towards building a universal defect prediction model”. In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by Premkumar T. Devanbu, Sung Kim and Martin Pinzger. ACM, 2014, pp. 182–191. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597078.

[170] Jie Zhang et al. “A survey on bug-report analysis”. In: *SCIENCE CHINA Information Sciences* 58.2 (2015), pp. 1–24. DOI: 10.1007/s11432-014-5241-2.

[171] Tong Zhang. “Solving large scale linear prediction problems using stochastic gradient descent algorithms”. In: *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*. Ed. by Carla E. Brodley. Vol. 69. ACM International Conference Proceeding Series. ACM, 2004. DOI: 10.1145/1015330.1015332.

[172] Jian Zhou, Hongyu Zhang and David Lo. “Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports”. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy and Mauro Pezzè. IEEE Computer Society, 2012, pp. 14–24. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227210.

[173] Thomas Zimmermann and Nachiappan Nagappan. “Predicting defects using network analysis on dependency graphs”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer and Volker Gruhn. ACM, 2008, pp. 531–540. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368161.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/