DR_BEV: Developer Recommendation Based on Executed Vocabulary

Alon Bendelac

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Francisco Servant, Chair

Na Meng

Osman Balci

May 12, 2020

Blacksburg, Virginia

Keywords: Bug Assignment, Bug Triage, Security Vulnerability, Information Extraction

DR_BEV: Developer Recommendation Based on Executed Vocabulary

Alon Bendelac

(ABSTRACT)

Bug-fixing makes up a large portion of software development expenses. Once a bug is discovered, it must be assigned to an appropriate developer who has the necessary expertise to fix the bug. While this bug-assignment task has traditionally been done manually, automatic bug assignment techniques have been developed to facilitate this task. Most of the existing techniques are report-based. That is, they work on bugs that are textually described in bug reports. However, only a subset of bugs that are observed as a faulty program execution are also described textually. Certain bugs, such as security vulnerability bugs, are only represented with a faulty program execution, and are not described textually. Promptly fixing these software security vulnerability bugs is necessary in order to manage security threats. Accordingly, execution-based bug assignment techniques, which model a bug with a faulty program execution, are an important tool in fixing software security bugs. In this thesis, we compare WhoseFault, an existing execution-based bug assignment technique, to report-based techniques. Additionally, we propose *DR_BEV* (***D***eveloper ***R***ecommendation ***B***ased on ***E***xecuted ***V***ocabulary), a novel execution-based technique that models developer expertise based on the vocabulary of each developer's source code contributions, and we demonstrate that this technique out-performs the current state-of-the-art execution-based technique. Our observations indicate that report-based techniques perform better than execution-based techniques, but not by a wide margin. Therefore, while a report-based technique should be used if a report exists for a bug, our results should provide confidence in the scenarios in which only execution-based techniques are applicable.

DR_BEV: Developer Recommendation Based on Executed Vocabulary

Alon Bendelac

(GENERAL AUDIENCE ABSTRACT)

Bug-fixing, or fixing known errors in computer software, makes up a large portion of software development expenses. Once a bug is discovered, it must be assigned to an appropriate developer who has the necessary expertise to fix the bug. This bug-assignment task has traditionally been done manually. However, this manual task is time-consuming, error-prone, and tedious. Therefore, automatic bug assignment techniques have been developed to facilitate this task. Most of the existing techniques are report-based. That is, they work on bugs that are textually described in bug reports. However, only a subset of bugs that are observed as a faulty program execution are also described textually. Certain bugs, such as security vulnerability bugs, are only represented with a faulty program execution, and are not described textually. In other words, these bugs are represented by a code coverage, which indicates which lines of source code have been executed in the faulty program execution. Promptly fixing these software security vulnerability bugs is necessary in order to manage security threats. Accordingly, execution-based bug assignment techniques, which model a bug with a faulty program execution, are an important tool in fixing software security bugs. In this thesis, we compare WhoseFault, an existing execution-based bug assignment technique, to report-based techniques. Additionally, we propose *DR_BEV* (***D***eveloper ***R***ecommendation ***B***ased on ***E***xecuted ***V***ocabulary), a novel execution-based technique that models developer expertise based on the vocabulary of each developer's source code contributions, and we demonstrate that this technique out-performs the current state-of-the-art execution-based technique.

Dedication

To my loving parents, Uri and Maly Bendelac.

Acknowledgments

I'd like to thank my research advisor, Dr. Francisco Servant, for his guidance. I would also like to thank Dr. Na Meng and Dr. Osman Balci for serving on my committee. I'd like to thank Dr. Patel and Dr. Grey for their immeasurable help. Last, but certainly not least, I'd like to thank my family. I'd like to thank my brother Noam for keeping me company during my last year in college, my sister Shiri for paving the way through the education system, my mother Maly for her endless emotional support, and my father Uri for pushing me beyond my limits.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bug-fixing makes up a large portion of software development expenses. A study [1] has found that in 2017, the cost of software bugs was over 1.7 trillion dollars. Another study [2] has found that 90% of software cost is maintenance-related. This problem also exists in software security. In a study [3], 90% of businesses surveyed admitted having a security incident, and the average cost of a security breach was over half-a-million dollars. Therefore, tools to facilitate the bug-fixing process in software development can help lower software maintenance costs.

Once a software bug is identified, it must be assigned to an appropriate developer who has the necessary expertise to fix the bug. The process of making this bug-to-developer assignment is known as *bug triage* [4]. Traditionally, bug triage has been done manually by a developer referred to as the *triager* [4]. This manual work, however, has the following three problems:

1. **Manual bug triage is time-consuming.** The Mozilla bug repository received an average of nearly 300 daily bug reports [4]. If the triager spent, on average, five minutes per bug, that adds up to 25 hours per day.

2. **Manual bug triage is error-prone.** Often, an erroneous bug-to-developer assignment, in which the assigned developer does *not* have the necessary expertise to fix the bug, is made. In such a case, the assigned developer reassigns the bug to a more suitable developer. This is known as *bug reassignment* or *bug tossing* [5]. In the Mozilla

and Eclipse projects, between 37% and 44% of bug reports are tossed at least once [6].

3. **Manual bug triage is tedious.** It requires keeping track of developers and their expertise, especially as the project grows and more components and developers are added. In the Eclipse project, for example, over 1,200 developers were assigned to multiple bug reports [6].

While bug triage has traditionally been done manually, automatic bug assignment techniques have been developed to facilitate this task. Given a bug, the objective of a bug assignment technique is to rank developers by descending expertise towards the bug. That is, the highest ranked developer is the one predicted to be most suitable to fix the bug.

Most of the existing bug assignment techniques are report-based. That is, they are designed for bugs that are textually described in a bug report, which, in turn, is managed in a bug repository. These techniques operate by mining the project's bug repository or code repository, and applying natural language processing techniques to compare the text in a new bug report to the text in those software artifacts. Because these techniques require a textual bug description, they work for bugs that a human user of the software system can observe and describe. Therefore, these techniques are often tested on bugs in user applications such as Eclipse and Mozilla products [7].

Only a subset of bugs that are observed as faulty code executions, however, are also described in a textual bug report. Certain bugs, such as security vulnerability bugs, are discovered with a faulty code execution, but are not textually described in a bug report [8]. Such security vulnerability bugs are not observed by the user, but rather, are identified via vulnerability detection techniques during security testing [9]. Such vulnerabilities include SQL injections [10, 11], buffer overflows [12], format-string attacks [13], and network attacks [14].

Promptly fixing these software security vulnerability bugs is necessary in order to manage

security threats. Accordingly, *execution*-based, as opposed to *report*-based, bug assignment techniques are an important tool in fixing software security bugs. To the best of our knowledge, the only existing execution-based technique is WhoseFault by Servant et al. [15]. Therefore, we consider WhoseFault to be the current state-of-the-art execution-based bug assignment technique.

This thesis has three goals.

The first goal of this thesis is to empirically determine whether there exist situations in which a failed execution alone, and *not* a bug report, is available for a bug. To the best of our knowledge, this is the first study to do so. The purpose of this goal is to motivate the necessity for effective execution-based bug assignment techniques. To achieve this goal, we conduct an empirical study of bug-fix commits in eight open-source software projects, and we investigate the prevalence of references to bug reports and failed code executions in those commits.

The second goal of this thesis is to empirically compare the performance of WhoseFault, an execution-based technique, to report-based techniques in order to determine if the execution-based technique provides results as accurate as the report-based techniques. While previous studies have compared report-based techniques, to the best of our knowledge, no study has compared execution-based and report-based techniques. To achieve this first goal, we have benchmarked several state-of-the-art bug assignment techniques on a dataset of bugs. We have constructed a dataset of 340 bugs from eight open-source software projects. We selected seven bug assignment techniques, and have benchmarked them in terms of effectiveness and efficiency. Our observations indicate that report-based techniques perform better than execution-based techniques, but not by a wide margin. Therefore, while a report-based technique should be used if a report exists for a bug, our results should provide confidence in the scenarios in which only execution-based techniques are applicable.

The third goal of this thesis is to provide a novel execution-based technique that is more accurate than the state-of-the-art execution-based technique. To the best of our knowledge, the only existing execution-based technique is WhoseFault [15], and therefore, we consider it to be the current state-of-the-art execution-based technique. To achieve this second goal, we propose *DR_BEV* (***D***eveloper ***R***ecommendation ***B***ased on ***E***xecuted ***V***ocabulary), a novel execution-based technique that is based on the vocabulary that developers use in their source code contributions. We benchmark this technique on the same bug dataset used for the first goal. Our results suggest that *DR_BEV* is both more effective and more efficient than WhoseFault, the current state-of-the-art execution-based technique.

In this thesis, we make the following research contributions:

1. We review existing bug assignment techniques. While other studies [5, 16, 17] have also reviewed existing techniques, those studies do not include all newer techniques. To the best of our knowledge, our literature review includes all existing techniques.

2. We conduct an empirical study of bug-fix commits in open-source software projects. Our objective is to determine whether there exist situations in which a failed execution alone, and *not* a bug report, is available for a bug.

3. We benchmark existing state-of-the-art bug assignment techniques on a dataset of bugs from eight open-source software projects. Our objective is to compare the performance of report-based and execution-based techniques in terms of effectiveness and efficiency.

4. We present *DR_BEV*, a novel execution-based bug assignment technique that does not require a textual bug report. This technique models developer expertise based on the vocabulary of each developer's source code contributions.

5. We evaluate *DR_BEV* on a dataset of bugs, and we compare its performance to that

of WhoseFault, the current state-of-the-art execution-based technique.

Chapter 2

Review of Literature

In this chapter, we provide some background related to automatic bug assignment. First, we describe two software artifacts, namely bug repositories and code repositories, that are commonly leveraged by bug assignment techniques. Second, in order to more concretely motivate execution-based bug assignment, we describe security vulnerability detection techniques that model bugs as faulty code executions. Finally, we summarize existing bug assignment techniques.

2.1 Software Artifacts

To understand the general methodologies behind existing bug assignment techniques, we describe two software artifacts, namely bug repositories and code repositories, that are commonly leveraged by existing bug assignment techniques.

A bug repository is managed by a Bug Tracking System (BTS), and is used to track bug reports. There are a number of BTSs, including Bugzilla [18], JIRA [19], and the GitHub issue tacking system [20]. Each bug report in a BTS has a number of fields. While these fields vary by BTS, Table 2.1 summarizes the most common fields.

A code repository is managed by a Version Control System (VCS), and is used to track revisions of the project's source code. There are a number of VCSs, including Git [21],

Table 2.1: Bug Report Field Descriptions

Field	**Type**	**Description**
Summary	Text	One-line summary of bug description
Description	Text	Free-text description of bug
Reported by	ID	ID of developer who reported the bug
Fixed by	ID	ID of developer who fixed the bug
Creation date	Timestamp	Date and time of when report was created
Fix date	Timestamp	Date and time of when report was marked as fixed
Status	Categorical	Whether the report is open or closed
Resolution	Categorical	Whether the report is fixed, a duplicate, won't be fixed, etc.
Product	Categorical	The product that is affected by the bug
Component	Categorical	The component within the product affected by the bug

Table 2.2: Commit Field Descriptions

Field	**Type**	**Description**
Author	ID	ID of developer who made the commit
Date	Timestamp	Date and time of when the commit was made
Commit Message	Text	Short description of code changes

Apache Subversion (SVN) [22], and Concurrent Versions System (CVS) [23]. Revisions of code in a VCS are commonly called *commits*. Table 2.2 summarizes the most important fields in commits.

2.2 Vulnerability Detection Techniques

There are many situations in which existing report-based bug assignment techniques are not helpful. One important example of such situations is when automated techniques detect security vulnerabilities that have to later be fixed by an expert software engineer. These techniques provide an execution of the vulnerability, but not a natural language report. In order to more concretely motivate execution-based bug assignment, we summarize a number of security vulnerability detection techniques that produce a faulty code execution trace.

Zhang et al. [24] proposed SecTAC, a vulnerability detection technique that is based on trace analysis and symbolic execution. In this technique, existing test methods are used to generate execution traces, which are sequences of source code statements exercised by a test case. Each execution trace is symbolically executed to determine if an assignment of values to variables violates a security constraint. An execution trace is flagged as having a security vulnerability if indeed an assignment of values to variables violates a security constraint. They tested their technique on C functions that are known to have security vulnerabilities such as buffer overflow and formatting string.

Li et al. [8] improved on SecTAC by using backward trace analysis. SecTAC cannot handle vulnerabilities that are not covered in the execution paths produced by the given test cases. Li et al.'s technique starts by finding hot spots, or security-sensitive functions, in the source code. Then, a data flow tree is constructed, and the possible execution traces that lead to the hot spots are identified. Lastly, symbolic execution is applied to these identified execution traces. If an execution trace leads to an assignment of values to program variables that violated a security constraint, then the hot spot corresponding to the execution trace is flagged as vulnerable.

Halfond et al. [25] developed a technique to detect SQL injection attacks. In such attacks, a malicious input string to an SQL query results in an illegal query to a database. First, static program analysis is used to build a model of legitimate queries. Then, runtime monitoring is applied to check a dynamically-generated query against the statically-built model. If an illegal query is detected, then the execution is reported as violating a security constraint.

Xu et al. [26] developed a probabilistic-based program anomaly detection technique. The technique computes the likelihood of an executed sequence of program calls to occur. A classifier, based on a probabilistic control-flow model, is trained to detect malicious code executions.

2.3 Bug Assignment Techniques

Next, we summarize existing bug assignment techniques. In order to summarize these existing techniques in an organized manner, we classify them into a hierarchy of families of similar techniques, as shown in Figure 2.1. This hierarchy consists of three levels. The first level distinguishes between execution-based and report-based techniques. The second level separates report-based techniques into machine learning and information retrieval families. Finally, at the third level, the report-based techniques are further grouped into families of similar approaches.

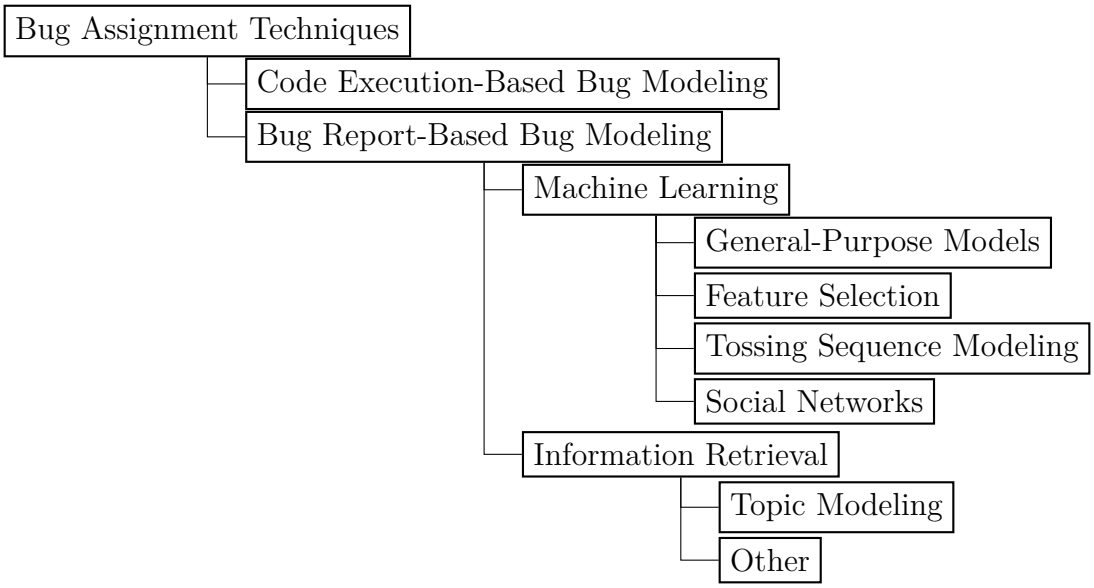


Figure 2.1: Hierarchical Classification of Bug Assignment Techniques

2.3.1 WhoseFault, an Execution-Based Technique

Servant et al. [15] developed WhoseFault, a bug assignment technique that is based on the execution of test cases. To model a bug, they use the Tarantula [27] fault localization technique, which computes, for each line of source code, a suspiciousness score based on the

line-level code coverage of passing and failing test cases. The higher a line's suspiciousness score, the higher the likelihood that this line contains a bug. A developer's expertise score is measured based on how suspicious the lines that they modified are, and how recent those modifications were. The idea is that developers who have recently modified highly suspicious lines of source code are likely to have the necessary expertise to fix the bug. To the best of our knowledge, this is the only existing execution-based bug assignment technique.

2.3.2 Report-Based Techniques: Machine Learning Models

In machine learning (ML) techniques, bug assignment is modeled as a supervised classification problem, in which developers are treated as classes and bug reports are treated as instances. The developer who fixed the bug is considered to be the ground truth class. Generally, these techniques follow three main phases: text processing, training, and inference. First, in the text processing phase, natural language processing techniques are used to preprocess and vectorize the textual descriptions in bug reports. Next, in the training phase, a supervised ML model is trained using the labeled instances (i.e. the fixed bug reports). Lastly, in the inference phase, the trained model is used to predict the most suitable developer for a new, unfixed bug report.

General-Purpose Models

This family of techniques apply general-purpose supervised ML classification models to bug assignment.

Cubranic et al. [28] pioneered the bug assignment problem by modeling it as a text classification problem. Their approach compares the text of a given bug report to the texts of previous bug reports. The idea of their approach is that the developer who fixed previous

bug reports whose texts are similar to a given bug report's text should be recommended to fix the bug. They modeled textual bug descriptions as Bag-of-Word vectors. They used Naive Bayes, a supervised ML model, to perform the text classification. The input to the model is the vectorized bug description, and the output of the model indicates which developer is predicted to be the most suitable to fix the bug.

Anvik et al. [4] extended the work of [28] in a number of ways. They used TF-IDF vectorization to model the textual bug descriptions. Their approach outputs a ranked list of recommended developers, as opposed to just one developer. They tested a number of supervised learning algorithms, and reported that the Support Vector Machine (SVM) algorithm performed the best. Lastly, they also cleaned their dataset of bug reports by filtering out reports that were fixed by inactive or low-activity developers.

Lin et al. [29] applied ML-based bug assignment on Chinese bug data. They developed two approaches that compared the use of textual and non-textual fields in bug reports. In the first approach, they applied ML-based bug assignment on the textual fields of bug reports with an SVM model. In the second approach, they explored the use of non-textual fields of bug reports with a decision tree model. In their experiment, the non-textual approach outperformed both the text-based approach as well as manual bug triage.

Recently, more advanced ML models have also been applied to the bug assignment domain. Jonsson et al. [30] used stacked generalization. In stacked generalization, several independent ML classifiers are trained, and an additional ML classifier uses those independent classifiers to make a final prediction. Lee et al. [31] deployed a Convolutional Neural Network (CNN), and vectorized textual bug descriptions with Word2Vec [32] word embeddings. Lastly, Mani et al. [33] deployed a Recurrent Neural Network (RNN). RNNs are designed for feature vectors that represent a temporal sequence, such as the sequence of words in a sentence or a document.

Feature Selection

This family of techniques apply feature selection methods on ML-based bug assignment techniques. The purpose of these feature selection methods is to reduce both the dimensionality and the sparseness of data in order to improve the accuracy of bug assignment.

Ahsan et al. [34] investigated the use of feature selection methods, namely term frequency and latent semantic indexing (LSI), to reduce the dimensionality of the feature space in an ML model. With the term frequency method, terms with a frequency below some threshold are ignored. The objective of LSI is to deal with the issues of synonymy and polysemy. Synonymy refers to different words with a similar meaning, and polysemy refers to words with multiple meanings that are inferred from the context. They tested seven ML models, and obtained the best results when using LSI with SVM.

Zou et al. [35] applied feature selection and instance selection techniques in order to deal with large-scale and low-quality bug datasets. They used the χ^2-test for feature selection and Iterative Case Filter for instance selection. When they removed 70% of words in bug reports and 50% of bug reports, they achieved higher accuracies than when using the original bug dataset in its entirety.

Alenezi et al. [36] investigated five term selection methods on an ML model. These methods select the most discriminating terms for the classification task. Their results suggest that the χ^2 term selection method outperforms the other four methods.

Wu et al. [5] used statistical methods to identify three factors that contribute to about 90% of variance in tossing path length (i.e. the number of times that a bug report is tossed among developers). Then, they applied these factors to improve ML-based bug assignment. They concluded that their approach can be used as a feature selection method for ML-based bug assignment.

Tossing Sequence Modeling

This family of techniques model tossing sequences as bug tossing graphs to improve ML-based bug assignment.

Jeong et al. [6] were the first to propose leveraging bug tossing graphs for bug assignment. To model bug tossing graphs, they used Markov chains, with states representing developers and transaction probabilities representing tossing probabilities. They used tossing graphs to improve ML-based bug assignment as follows. First, they use an ML approach to generate a list of recommended developers. Then, for each such recommended developer, they use the tossing graph to determine to which developer is the recommended developer most likely to toss the bug to. These new developers are then augmented into the recommendation list. In their experiments, their model has both increased the accuracy of bug triage and decreased the number of bug tosses.

Bhattacharya et al. [37] implemented multi-feature tossing graphs in which edges are labeled with developer expertise, and nodes are labeled with developer activity. In order to label graph edges with developer expertise, they label edges with the product and component affiliated with the tossed bug. In order to label graph nodes with developer activity, they label nodes with the difference between the date of the bug being validated and the date of the last activity of that developer. Like [6], they determine which developers have a tossing relationship with the ML-based recommended developers, and then augmented these new developers into the recommendation list.

Wang et al. [38] expanded the idea of a bug tossing graph from a homogeneous network to a heterogeneous network. In their heterogeneous network, there are multiple types of nodes, namely developers, bugs, comments, components, and products. There are also multiple types of edges to denote different relations among nodes. First, they use an ML model to

predict recommended developers. Then, for each such recommended developer, they use the heterogeneous network to determine the developer who is most likely to collaborate with the recommended developer. Finally, they augment these new developers to the recommendation list.

Zhang et al. [39] have also used the idea of heterogeneous networks. In their approach, they first used K-Nearest-Neighbor (KNN) to find bug reports similar to the new bug report. Then, they use heterogeneous proximity to rank the developers who contributed to those similar bug reports. Their heterogeneous proximity approach is based on counting the number of connections between entities in the network.

Xi et al. [40] developed a sequence-to-sequence model that is trained on both the textual description as well as the tossing sequence of previous bug reports. The model predicts the tossing sequence, which begins with the bug report reporter, for the given bug report. The last developer in the predicted tossing sequence is predicted to be the most suitable to fix the bug, and the other developers in the predicted tossing sequence are considered as additional relevant developers.

Huang et al. [41] proposed the use of a multiplex, as opposed to a single-layer, tossing network. They have built a collaborative multiplex network consisting of a tossing graph and an email communication graph. They processed the data in the multiplex network into vectors, which were used to train an ML model.

Xi et al. [42] were the first to combine textual description, metadata, and tossing sequence for bug triage. First, a sequence-to-sequence model learns textual description and tossing sequence; and then, a classification model integrates the three information sources.

Social Networks

This family of techniques leverage social network models to rank developers that are recommended with an ML model.

Wu et al. [43] used K-Nearest-Neighbors (KNN) to search for bug reports similar to a given bug report, and then used frequency and social network metrics to rank developer expertise based on developer participation in discussion threads in those similar bug reports. They achieved the best results using their Out-Degree and Frequency techniques. The Frequency technique counts each developer's participation in the comments of the similar bug reports. The Out-Degree score is computed from a social network graph in which developers are modeled as nodes and edge weights are computed based on cosine similarities of developers that participated in the bug reports' discussions.

Xuan et al. [44] improved ML-based bug assignment with a developer prioritization technique. The developer prioritization technique is based on a social network in which developers represent nodes and comment counts represent edges. They used SVM and Naive Bayes as ML models to predict a list of relevant developers. Then, they rank these relevant developers by descending developer prioritization scores. Their experiments indicated that applying the developer prioritization technique to an ML model has increased bug triage accuracy over a stand-alone ML model.

2.3.3 Report-Based Techniques: Information Retrieval Models

In Information Retrieval (IR) techniques, developers are treated as documents and an incoming bug report is treated as a search query. Given a search query, the objective is to retrieve the correct document. That is, given a new bug report, the objective is to retrieve the developer who has the necessary expertise to fix the bug. IR techniques are based on

indexing bug reports or computing similarity between bug reports.

Topic Modeling

These techniques used topic modeling. A commonly used topic modeling technique is Latent Dirichlet Allocation (LDA). LDA models documents as collection of topics, and it models topics as collections of words.

Park et al. [45] developed a technique that considers both accuracy and cost. They define cost as the time it takes a particular developer to fix a given bug. They use LDA to group bugs into categories, and they estimate each developer's cost towards each bug category. For each developer, and for each bug type, they estimate the cost of having the developer fix a bug of that type based on the previous costs of that developer fixing bugs of that type, and they rank developers by cost. They use an ML-model to rank developers by accuracy, and then they merge the accuracy-based and the cost-based rankings.

Xie et al. [46] used LDA topic modeling to group fixed bug reports into topics. Next, a probabilistic model is used to score a developer's expertise towards a new bug as a function of the bug's affiliation to each topic and the developer's expertise towards the topics. Finally, the developers are ranked by descending expertise scores.

Naguib et al. [47] used LDA topic modeling to categorize bug reports into topics. Then, the bug repository was mined to create developer profiles. The set of topics that model the new bug report were determined. Developers who have expertise in at least half of the bug report's topics are considered relevant. To rank these relevant developers, a score was computed for each developer based on their activity profile.

Xia et al. [48] developed a technique that performs bug report-based (BR-based) and developer-based (D-based) analyses. In the BR-based analysis, bug reports similar to the new bug

report are found by performing feature engineering that including NLP and LDA, and then using a multi-label ML model to predict relevant developers. In the D-based analysis, each developer's affinity towards the new bug report is measured based on feature engineering, including NLP and LDA, on previous bug reports fixed by the developer. A linear combination is used to combine the BR-based and the D-based analyses and produce a final recommendation list.

Yang et al. [49] first use LDA topic modeling to group bug reports. Given a new bug report, the subset of related topics to the report are determined, and the participating developers of the reports related to the topics are extracted. Also, the bug reports that have the same product, component, priority and severity as the given bug are determined, and the participating developers are extracted. These developers are ranked based on the number of assignments they received, the number of attachments in the reports assigned to them, the number of commits made by them, and the number of comments posted by them.

Xia et al. [50] proposed multi-feature topic modeling as an extension of LDA. It takes into account categorical features, namely product and component, in addition to the textual description of the bug. It is trained on old bug reports and inference is done on the new bug report to compute a distribution of topic scores, putting special emphasis on bug reports of the same feature combination as the new bug report's feature combination. An affinity score is computed for each developer, and the developers are ranked by descending affinity score.

Other

Canfora et al. [51] constructed a probabilistic IR model for bug assignment. They mine bug descriptions from Bugzilla and commit messages from CVS. Then, they index developers and source files as documents in an information retrieval system. They use these indexes to

recommend a developer to fix a bug, and to predict the source files that contain the bug.

Matter et al. [52] developed a vector space model (VSM) approach. They model each developer's expertise, as well as the query bug report, as a vector, and they use cosine similarity to rank developers based on the similarity between their expertise vector and the query vector. Their vectorization technique is based on a Bag-of-Words (BoW) model. First, they mine the code repository to retrieve each developer's source code contributions. Then, they vectorize each such contribution into a BoW vector, and each of these vectors is weighted by the recency of the contribution. For each developer, their contribution vectors are aggregated. Lastly, each developer expertise vector is weighted by the recency of the developer's most recent contribution to the code repository.

Helming et al. [53] used a dataset in which work items are linked to functional requirements. Given a new work item, they identify related work items, and score developers based on the number of those related work items that they completed. Their model is designed for work items such as bug reports and other tasks.

Tamrawi et al. [54] implemented a fuzzy set model for developer expertise. Their approach is based on the idea that each textual bug description is made of a set of technical terms, and when a developer fixes a bug, they demonstrate expertise related to these terms. For each term, a fuzzy set is constructed to represent how much expertise each developer has towards that term. Then, a membership function scores a developer's expertise towards the terms in a textual bug description, and developers are ranked by descending expertise scores.

Kagdi et al. [55] developed a heuristic-based technique to recommend developers based on developer contributions in the code repository. The premise of their approach is that developers who have contributed to suspicious files are more likely to have the expertise to fix a given bug. They use LSI to locate and rank units of source code relevant to a bug report.

Then, they mine these source code units in the code repository to rank the developers who have contributed to these source code units.

Linares-Vasquez et al. [56] developed an IR-based bug assignment technique that does not require mining of either a code repository or a bug repository. The technique takes advantage of code authorship information available in the header comments of source code files. LSI is used to index source code entities. To retrieve relevant source code entities, document similarity is computed between a textual bug description, used as a query, and the source code entities in the index. Then, the authorship information in those relevant source code entities are used to recommend a list of developers.

Nagwani et al. [57] modeled developer expertise based on the frequent terms in the bug reports that each developer has fixed. For each developer, a list of frequent terms in the bug reports that they fixed is generated. Then, given a new bug report, developer expertise is measured based on the similarity between the new bug report terms and each developer's list of frequent terms.

Shokripour et al. [58] proposed a two-phased approach that uses both the bug repository and the code repository. In the first phase, the buggy source code files are predicted. To do so, they use four information sources: identifiers, commit messages, comments in source code, and reports of previously fixed bugs. They parse the bug report text and the source code to create an index of unigram nouns. During the second phase, they recommend developers based on information about who has previously fixed faults in the predicted source code files.

Hossen et al [59] developed a technique that uses change proneness information to improve automatic bug assignment. Change proneness is a measure of its change affinity, derived from each source code entity's change history. They identify buggy source code units by comparing the textual bug description to the source code. Then, they use change proneness

to rank the buggy source code units. Lastly, they rank developers based on their activity with those ranked buggy source code units.

Hu et al. [60] developed an approach that uses a Developer-Component-Bug (DCB) network. In a DCB network, developers are linked to the project components that they have worked on, and bugs are linked to the project components that they are fixed in. For a new bug report, they use the vector space model (VSM) to compute the similarity between the new bug and old bugs. Then, they use the DCB network to calculate relevance between the new bug and the developers.

Badashian et al. [61] used Q&A platform activity to model developer expertise. Their approach uses answers in the Q&A platform as evidence of expertise, and they use the number of up-votes of each answers to measure the extent of expertise. They use the tags on questions to model technical components of possible expertise. They used Stack Overflow as the Q&A platform in their experiments.

Zanjani et al. [62] used developer-IDE interactions to model developer expertise. Developer interactions with the IDE include navigate, view, and modify. They first employ an ML technique to locate source code entities relevant to the textual bug description. Then, the interaction histories of these entities are mined to recommend developers.

Yang et al. [63] developed a technique that recommends relevant files regarding the new bug in addition to recommending relevant developers. To identify relevant commits, they measure term similarity between the new bug report and the historical commits using the cosine function. They use collaborative topic modeling (CTM) to identify personalized files for the recommended developers.

Peng et al. [64] developed a relevant search technique. They use an inverted index to map a term to the list of documents that contain the term. They use this index to search for bug

reports that are similar to the new bug report. Then, developers are scored based on the similarity of the similar bug reports to the new bug report.

Chapter 3

Empirical Study of Bug-Fix Commits

Most of the existing techniques in the bug assignment domain are designed for cases in which a bug report is available. However, some studies have indicated situations in which only a failed code execution may be available. A few examples include security vulnerability detection techniques that flag code executions that cause potential security vulnerabilities [26]; failures in continuous integration pipelines that are captured in failing test cases [65]; and bugs found in crowdsourced mobile testing, in which executed bugs are often reported with screenshots [66].

In this chapter, we perform an empirical study of bug-fix commits, or commits that fix bugs, in the code repositories of eight open-source software projects. The objective of this study is to empirically investigate whether there exist situations in which a failed code execution alone, and *not* a bug report, is available for a bug. These potential situations, in turn, are the motivation for execution-based bug assignment techniques.

In this chapter, we seek to answer the following research question:

1. How prevalent are bug reports and failed code executions?

3.1 Research Method

We studied eight open-source software projects: JFreeChart, Commons Lang, Commons Math, Mockito, Joda-Time, Commons IO, Rhino, and AspectJ. These projects are well-known and commonly used in the software engineering domain. We mined and analyzed the commits in each project's code repositories. From the collection of commits, we identified the bug-fix commits. Then, we counted the number of those bug-fix commits that reference both a bug report and a failed execution, only a bug report, only a failed execution, and neither a bug report nor a failed execution.

In order to make these identifications, we developed the following heuristic:

- A commit is a *bug-fix commit* if the commit message contains the word "fix".
- A bug-fix commit has a corresponding *bug report* if the commit message contains a reference to a bug report ID.
- A bug-fix commit has a corresponding *failed execution* if the commit message or the filename of one of the modified files contains the word "test".

Note that while this heuristic is based upon reasonable assumptions, it is *not* intended to be a perfect heuristic. However, the objective of this study is not necessarily to compute exact numbers, but rather, the objective is to gain a general understanding of whether there exist bugs *without* a bug report, but *with* an execution.

3.2 Results

The results of this experiment are shown in Table 3.1. In this table, the columns indicate the following information:

- ***Project***: Name of software project.
- ***All***: Number of commits in the code repository.
- ***Bug-Fix***: Number of bug-fix commits.
- ***Both***: Number of bug-fix commits with both a bug report and a failed execution.
- ***Report***: Number of bug-fix commits with a bug report, but without a failed execution.
- ***Execution***: Number of bug-fix commits without a bug report, but with a failed execution.
- ***Neither***: Number of bug-fix commits with neither a bug report nor a failed execution.

Additionally, the last two rows of the table indicate the following information. In the *All Projects* row, we sum the counts for all projects combined. In the *% of Bug-Fix* row, we compute each count from the *All Projects* row as a percentage of the number of bug-fix commits in all projects combined.

As shown in the last row of the table, *report without execution* was the most common situation, at 33.67%, and *execution without report* was the least common situation, at 14.38%.

3.2.1 Discussion

The results of this experiment lead to the following observations. Bug reports are more prevalent than failed code executions. However, there exist situations in which only a failed code execution is available. In our experiment, approximately 14% of bug-fix commits had a reference to a failed code execution only, and not to a bug report. As a result, we conclude that there is an important need for effective execution-based bug assignment techniques.

Table 3.1: Frequency of Bug Reports and Faulty Executions in Bug-Fix Commits

Project	All	Bug-Fix	Both	Report	Execution	Neither
Commons IO	1538	205	45	13	54	93
Rhino	2242	828	14	436	40	338
AspectJ	12066	2496	994	585	424	493
JFreeChart	937	63	3	0	60	0
Commons Lang	3637	511	187	324	0	0
Commons Math	4966	831	298	533	0	0
Mockito	3188	597	183	64	164	186
Joda-Time	1710	337	19	21	102	195
All Projects	30284	5868	1743	1976	844	1305
% of Bug-Fix		100%	29.70%	33.67%	14.38%	22.24%

This is the key motivation for the next experiment, in which we benchmark existing state-of-the-art bug assignment techniques.

Chapter 4

Benchmarking Existing Techniques

Benchmarks allow for evaluation and comparison of systems or techniques based on factors such as effectiveness and efficiency. In this chapter, we perform a benchmark that compares WhoseFault, the existing execution-based bug assignment technique, to a number of state-of-the-art report-based techniques. The objective is to investigate how well the execution-based technique can perform compared to report-based techniques. While other studies have compared report-based techniques, to the best of our knowledge, no study has compared execution-based and report-based techniques.

In this benchmarking experiment, we seek to describe the relative performance of execution-based and report-based techniques as one of the following three possible cases:

- *Case 1: Execution-based techniques are worse than report-based techniques.* In this case, one should consider the option of writing a bug report. If a high-quality report can be efficiently written, then it makes sense to write one in order to make use of report-based techniques. Otherwise, execution-based techniques are the best available option.

- *Case 2: Execution-based techniques are better than report-based techniques.* In this case, we should transition to apply them in *all* scenarios in which a buggy execution exists.

- *Case 3: Execution-based and report-based techniques perform similarly.* In this case,

execution-based techniques could be beneficial because the human effort of writing a report could be avoided. Alternatively, a "hybrid" technique, which would take into account both kinds of input, could be developed.

In this chapter, we seek to answer the following research questions:

2. How does WhoseFault compare to report-based techniques in terms of *effectiveness*?

3. How does WhoseFault compare to report-based techniques in terms of *efficiency*?

4.1 Research Method

We selected a set of seven bug assignment techniques: one execution-based technique and six report-based techniques. We constructed a dataset of 340 bugs from eight open-source software projects. We ran each technique on each bug in our dataset. To measure effectiveness, we used the Normalized Discounted Cumulative Gain (NDCG) metric. To measure efficiency, we measured the execution time of each technique on each of the studied bugs. We evaluated these results to determine which one of the three cases, described at the beginning of this chapter, is observed in this benchmarking experiment.

4.1.1 Studied Techniques

We perform our benchmark on seven existing bug assignment techniques summarized in Table 4.1. Our rationale for selecting these seven techniques is as follows. We chose WhoseFault because, to the best of our knowledge, it is the only existing execution-based bug assignment technique. We chose Xia and Lee because they are the most recently published

bug-assignment techniques in the most selective software engineering journals and conferences. More specifically, they were published in Transactions on Software Engineering and ESEC/FSE, respectively. Lastly, we chose the other four techniques because they are some of the most seminal works in this area. To the best of our knowledge, we are studying more techniques than any other bug assignment evaluation did, as of the time of writing.

Table 4.1: Summary of Benchmarked Bug Assignment Techniques

Technique	**Expertise Modeling**	**Bug Modeling**	**Text Vectorization**	**Scoring Methodology**
Cubranic [28]	Fixed bug reports	New bug report	Bag-of-Words	Naive Bayes
Anvik [4]	Fixed bug reports	New bug report	TF-IDF	SVM
Matter [52]	Source code contributions	New bug report	Bag-of-Words	Cosine similarity
Tamrawi [54]	Fixed bug reports	New bug report	Bag-of-Words	Fuzzy sets
Lee [31]	Fixed bug reports	New bug report	Word2Vec	Convolutional Neural Network
Xia [50]	Fixed bug reports	New bug report	Bag-of-Words	Topic Modeling
WhoseFault [15]	Code repository mining	Fault localization		Heuristic

To further motivate the selection of each of these techniques, we provide a brief summary of each selected technique:

- ***Cubranic***: Cubranic et al. [28] model their technique as a supervised ML model, and they apply bag-of-words vectorization and a Naive Bayes classifier. In their work, they have not compared their technique to any baseline, since they have pioneered bug assignment. To the best of our knowledge, this is the second most-cited technique as of the time of writing.

- ***Anvik***: Anvik et al. [4] also model their technique as a supervised ML model, but

they apply TF-IDF vectorization and an SVM classifier. They have not compared their technique to any baseline. To the best of our knowledge, this is the most-cited technique as of the time of writing.

- ***Matter***: Matter et al. [52] model developer expertise based on the vocabulary of each developer's source code contributions, and they apply cosine similarity to score expertise towards a bug report. They have not compared their technique to any baseline. To the best of our knowledge, this is, as of the time of writing, the most-cited technique that models developer expertise from the history of the source code.

- ***Tamrawi***: Tamrawi et al. [54] use fuzzy sets to model developer expertise. They evaluated their technique against SVM, decision tree, and Naive Bayes ML classifiers. Tamrawi et al.'s [54] work has been used for comparison in a number of studies [39, 48, 50].

- ***Lee***: Lee et al. [31] apply Word2Vec word embeddings to vectorize bug descriptions, and they use a Convolutional Neural Network as a supervised ML model. They evaluated their techniques against manual bug triage.

- ***Xia***: Xia et al. [50] use LDA topic modeling to model developer expertise. They evaluated their technique against Tamrawi's technique.

- ***WhoseFault***: Servant et al. [15] developed WhoseFault, an execution-based technique. WhoseFault recommends developers who have made recent modifications to source code lines that have high fault localization suspiciousness values. They have evaluated their technique against naive simplifications of their proposed approach. To the best of our knowledge, WhoseFault is currently the only existing execution-based technique.

We used the original implementation of WhoseFault, and we implemented the other tech-

niques in Python using scikit-learn [67] for machine learning, TensorFlow [68] and Keras [69] for Convolutional Neural Networks, NLTK [70] for natural language processing, and PyDriller [71] for mining Git repositories.

4.1.2 Studied Bugs

Previous studies have shown that the performance of bug assignment techniques vary strongly between different projects. For example, Anvik et al. [4] evaluated their technique on three projects: Eclipse, Firefox, and GCC. Their technique performed significantly better on the Eclipse and Firefox projects than on the GCC project. Therefore, it is better to test bug assignment techniques on multiple projects than it is on one project.

In order to benchmark the bug assignment techniques, we need a dataset of fixed bugs. For each such bug in our dataset, we need the following data:

1. **Code repository**: We need the project's code repository, which will be mined by WhoseFault and Matter. In addition, we need to identify the post-fix commit - that is, the commit that contains the fix for the bug. Specifically, we need to determine who made that bug-fixing commit. Lastly, for WhoseFault, we need fault localization data (i.e. suspiciousness value for each executable line of code) for the pre-fix commit - that is, the commit directly before the post-fix commit. Note, the prefix-commit contains the bug, and the post-fix commit does not.

2. **Bug repository**: We need the project's bug repository, which will be mined by the bug report-based techniques. More specifically, we need the bug reports corresponding to the bugs in our dataset. Additionally, we need all the bug reports that were previously fixed to be used as the training data.

Table 4.2: Dataset of Studied Bugs

Source	Project Name	Identifier	Number of Bugs	Bug Repository
Defects4J	JFreeChart	Chart	7	Sourceforge
	Commons Lang	Lang	63	JIRA
	Commons Math	Math	103	JIRA
	Mockito	Mockito	38	GitHub
	Joda-Time	Time	23	Sourceforge, GitHub
Additional	Commons IO	IO	19	JIRA
iBugs	Rhino	Rhino	15	Bugzilla
	AspectJ	AspectJ	72	Bugzilla
		Total	340	

While we wanted to use a dataset of security vulnerability bugs, we were unaware of any such dataset that contains all the information needed to run this experiment. Therefore, we studied bugs in software engineering in general. We constructed a dataset of 340 fixed bugs from eight open-source software projects. This dataset is summarized in Table 4.2. For each of these bugs, the pre-fix commit contains at least one failing test case that demonstrates the bug. As shown in the *Source* column of the table, we used Defects4J [72] and iBugs [73], two datasets of real software bugs, to collect bugs from five and two software projects, respectively. Additionally, we mined Apache Commons IO to complement those two datasets. For brevity, we will refer to these projects by the short identifying names specified in the *Identifier* column of the table. To the best of our knowledge, we are studying more software projects than any other bug assignment evaluation did, as of the time of writing.

For each project in the dataset, we obtained fault localization data and mined the project's bug repository. For the Defects4J projects, we used the fault localization data provided by Pearson et al. [74]. For the other projects, we used a propriety dataset of fault localization data. To mine the bug repositories, we used an API for each of the four bug tracking systems.

We did not use all bugs in the Defects4J dataset for the following reasons:

- *Chart*: We have not used 19 bugs provided by Defects4J, because Defects4J did not

provide a link to a bug report for those bugs.

- *Lang*: We did not use 2 bugs provided by Defects4J because they were not marked as fixed in the bug repository.

- *Math*: We did not use 3 bugs provided by Defects4J because two of them were not marked as fixed in the bug repository, and one of them was the first bug to be marked as fixed in the bug repository (and therefore, there were no fixed reports to train on).

- *Time*: We did not use 4 bugs provided by Defects4J because they were not marked as fixed in the bug repository.

- *Closure*: Additionally, Defects4J includes bugs for the Closure Compiler project. We did not use any of those bugs because there are no publicly available bug reports for those bugs. Specifically, Google, which developed this project, has initially used a proprietary bug tracking system [**?**], and eventually transferred to using GitHub as a bug tracking system. The bug reports written in the old bug repository, which contained the necessary reports, were not transferred to the new repository.

4.1.3 Definition of Ground Truth

In the bug assignment domain, the ground truth is the set of developers who actually have the necessary expertise to fix a given bug. Other studies have defined slightly different variations of the ground truth. In this study, in order to be able to compare the performance of different techniques, we define the ground truth as a union of the definitions used by any of our studied techniques. We define the ground truth as the developer who made the bug-fixing commit in the code repository and also the developer who marked the bug report as fixed in the bug repository. These may or may not be the same developer, and therefore,

the ground truth consists of either one or two developers.

In 68% of the bugs in our dataset, the developer who made the bug-fixing commit has also marked the bug report as fixed (i.e. the ground truth consists of *a single* developer). In the other 32% of bugs, the developer who made the bug-fixing commit was not the developer who marked the bug report as fixed (i.e. the ground truth consists of *two* developers).

Mapping Developer Identifiers in Software Artifacts

Often, a single developer uses multiple identifiers across software artifacts or within a single software artifact. For example, a developer might use a username as one identifier and a full name as another identifier. Grouping developer identifiers that belong to the same human developer is an important task for correctly modeling developer expertise based on developer activities in software artifacts.

To address this task, we manually group developer identifiers that refer to the same developer. Then, we assign each developer a unique numerical identifier, and map all of a developer's identifiers to the developer's unique numerical identifier. This way, we avoid treating a single developer, with multiple identifiers, as multiple distinct developers.

Figure 4.1 shows a simple example of this mapping task. Identifiers from the code and bug repositories are on the left and right sides, respectively. The unique numerical developer identifiers are in the center of the figure. In this example, every developer used a single identifier in the bug repository. So, each identifier in the bug repository points to a distinct numerical identifier. Next, consider the identifiers in the code repository. Alice and Bob each used two distinct identifiers in the code repository. Alice's two code repository identifiers point to the same numerical identifier, 0. Likewise, Bob's two code repository identifiers point to the same numerical identifier, 1.

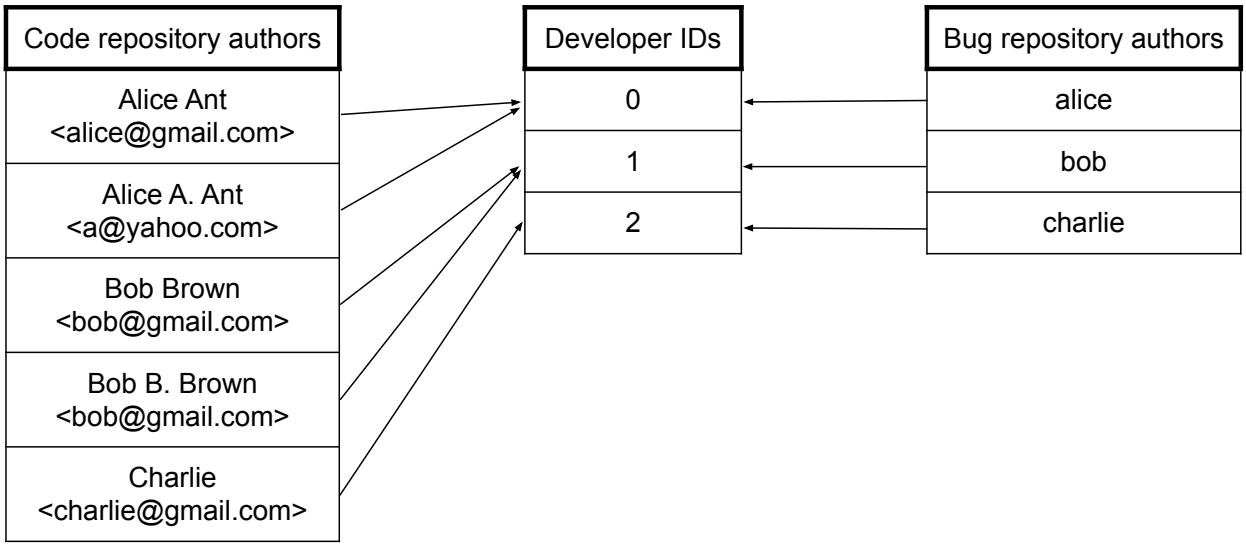


Figure 4.1: Example of Mapping Developer Identifiers in Software Artifacts

4.1.4 Effectiveness Evaluation Metric

It is important that we select an effectiveness evaluation metric that well suits our study. In this study, the evaluation metric should be able to score a ranking of items, or in this case, developers. An ideal ranking would have all the ground truth developers at the top of the ranked list. So, we want the evaluation metric to give a higher score the higher the ground truth developers are ranked, and a lower score the lower the ground truth developers are ranked. While precision and recall have been used in previous bug assignment studies, we argue that the Normalized Discounted Cumulative Gain (NDCG) metric is a more suitable evaluation metric for a ranking task.

Normalized Discounted Cumulative Gain

The Normalized Discounted Cumulative Gain (NDCG) metric is a measure of ranking quality [75]. NDCG is a normalization of DCG, which is based on the principle that relevant items are more useful the higher they appear in the ranking. In DCG and NDCG, each

item has a relevancy value between 0 and 1.0, and an ideal ranking would rank all items by descending relevancy. DCG is defined in Equation 4.1. In this equation, k is the number of top elements, in the recommendation ranking, that are considered, and rel_i is the relevance of the i^{th} element in the recommendation ranking.

$$\text{DCG@}k = \sum_{i=1}^{k} \frac{2^{\text{rel}_i} - 1}{\log_2(i+1)} \tag{4.1}$$

We use Burges et al.'s [76] definition of NDCG, shown in Equation 4.2, as the ratio between the DCG scores of the actual recommendation ranking and the ideal ranking. With this definition, an NDCG score is between 0 and 1.0, with a higher score indicating a better ranking. In our work, for a given bug, we assign a relevancy score of 1.0 to the ground truth developers, and a relevancy score of 0 to all other developers.

$$\text{NDCG@}k = \frac{\text{DCG@}k\text{(actual recommendation)}}{\text{DCG@}k\text{(ideal recommendation)}} \tag{4.2}$$

Comparison of Precision, Recall, and NDCG

Precision and recall have been commonly used for evaluation in previous bug assignment studies [4, 28, 34, 35, 36, 43, 51, 52, 56, 59, 60, 63, 64]. Precision is defined as the proportion of relevant items retrieved to total items retrieved. Recall is defined as the proportion of relevant items retrieved to total relevant items. Precision@k and recall@k only consider the top-k ranked items. Equations 4.3 and 4.4 define precision and recall, respectively, in the context of bug assignment.

$$\text{Precision} = \frac{\text{\# of appropriate recommendations}}{\text{\# of recommendations made}} \tag{4.3}$$

Table 4.3: Comparison of Precision, Recall, and NDCG

	Technique A	**Technique B**	**Technique C**
Ranking	***(1) Alice*** (2) Bob (3) Charlie	(1) Bob ***(2) Alice*** (3) Charlie	(1) Charlie (2) Bob ***(3) Alice***
Precision@3	0.33	0.33	0.33
Recall@3	1.00	1.00	1.00
NDCG@3	1.00	0.63	0.50

$$\text{Recall} = \frac{\text{\# of appropriate recommendations}}{\text{\# of possibly relevant developers}} \tag{4.4}$$

In Table 4.3, we compare precision, recall, and NDCG with a simple ranking task. In this example, there are three bug assignment techniques, labeled A, B, and C. The ground truth developer, as shown in bold and italics, is Alice. Techniques A, B, and C have ranked Alice in first, second, and third place, respectively. Therefore, an ideal evaluation metric would score techniques A, B, and C in descending order. We evaluated each technique using precision, recall, and NDCG at $k = 3$. As shown in the table, neither precision nor recall meet the requirement that techniques A, B, and C would be scored in descending order. NDCG, on the other hand, does meet this requirement. The reasoning behind this result is that precision and recall are designed for an *unordered selection* problem, while NDCG is designed for an *ordered ranking* problem.

So, we decide to use NDCG as our effectiveness evaluation metric. We evaluate our results using NDCG@1, NDCG@5, and NDCG@10. That is, we evaluate the single top recommendation, the top-5 recommendations, and the top-10 recommendations, respectively.

4.1.5 Efficiency Evaluation Metric

To measure the efficiency of each technique, we measured the execution time, in seconds, of each technique on each bug in our dataset.

4.1.6 Experiment Setup

In real-world use, when a bug report is being triaged, previously fixed bug reports can be used for triage. However, bug reports that haven't been fixed yet, or haven't even been written yet, cannot be used for triage. Therefore, each bug has a different training set of fixed bug reports. To accurately simulate this, we train and test on each bug in our dataset separately. For each bug in the testing set (see Table 4.2), the training set consists of all bug reports that have been fixed before the testing bug was fixed. The reasoning for comparing fix dates is as follows. Between the creation timestamp of the testing bug report and its fix timestamp, there could be other bug reports that get marked as fixed, which would increase the training set for this particular testing bug. So, during this time window, a bug assignment technique could be redeployed in order to include the newly fixed bug reports.

4.2 Results

4.2.1 Effectiveness

We ran each of the seven benchmarked techniques on our dataset of 340 bugs, and measured effectiveness using NDCG@1, NDCG@5, and NDCG@10. The results are shown in Table 4.4, in Figure 4.2 as bar charts and in Figure 4.3 as box plots. In each of those two figures, we have grouped the bugs by the eight software projects. Additionally, in the *All Projects* group,

results are shown for all 340 bugs as one project. In the box plots of Figure 4.3, we also show an *All Projects (Medians)* group. In this group, for each technique, the box plot shows eight median scores - one for each project.

There is a high variance in NDCG scores across projects. As seen in Figure 4.2, the highest scores were received on projects Chart and Time. These two projects have a small number of developers, which makes bug assignment easier. In project Chart, only four developers have fixed a bug. In project Time, 89% of bug reports have been marked as fix by a single developer. For project Chart, all techniques have received a perfect score of 1.0 at all three levels of NDCG. In project Time, Cubranic, Anvik, Lee, and Xia have received a perfect score, also at all three levels of NDCG.

Next, we consider the performance of Xia. The Xia technique takes into account the product and component categorical variables of bug reports. The projects that we test on use different bug tracking systems, some of which do and some of which do not support product and component fields. For those bug reports that do not have a product or component, we assigned a single dummy value for those fields. Four projects, namely Chart, Math, Mockito, and Time, have a single product-component pair each. Project IO has four product-component pairs, Rhino has three pairs, and Lang and AspectJ each has fourteen pairs. We would expect that Xia would perform better on projects with multiple product-component pairs, since that gives valuable information for the technique in order to "compartmentalize" bugs by product-component pairs. Indeed, at both NDCG@5 and NDCG@10, Xia was more accurate than all other techniques on projects Lang and AspectJ, which have the most product-component pairs.

In project Rhino, Matter received an exceptionally higher score than other techniques at all levels of NDCG. This could be explained by the fact that most of the textual bug descriptions in this project have included either some source code or a stack trace. Since Matter compares

the textual bug description to developers' source code contributions in the code repository, it makes sense that having source code in the bug description would boost Matter's score.

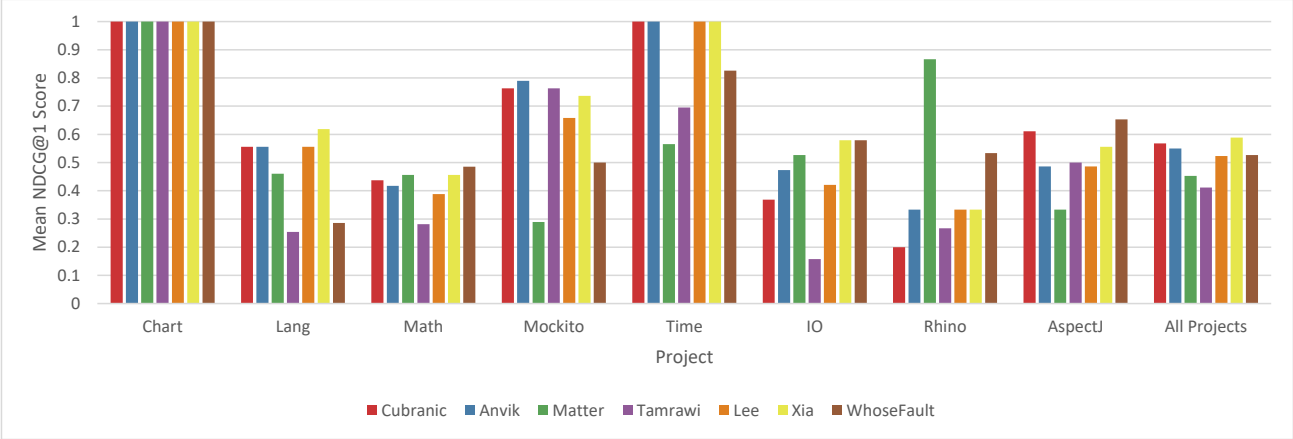
Overall, the most effective technique was Xia. At all three levels of NDCG, Xia received the highest mean score in the *All Projects* group.

Next, we consider the effectiveness of WhoseFault, the execution-based technique, compared to the effectiveness of the report-based techniques. At NDCG@1, WhoseFault has performed better than Matter, Tamrawi, and Lee, but worse than Cubranic, Anvik, and Xia. At NDCG@5, WhoseFault performed better than Tamrawi, but worse than all other five techniques. At NDCG@10, all six bug report-based techniques have performed better than WhoseFault.

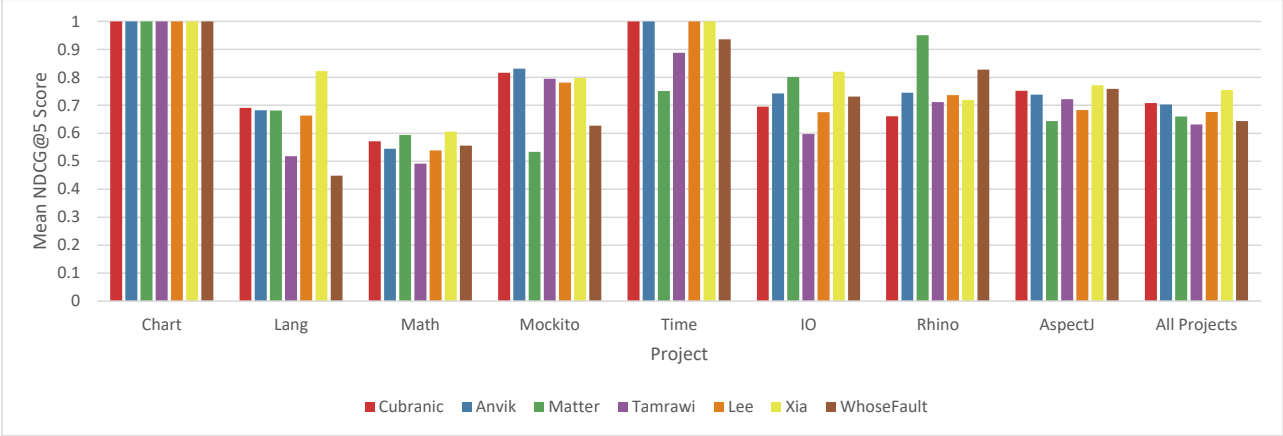
We want to test the statistical significance of the differences in scores between WhoseFault and the report-based techniques. To do so, we use the Wilcoxon signed-rank significance test. We chose the Wilcoxon statistical significance test due to the paired, or matched, nature of our data. That is, each technique was tested on the same dataset of bugs. In Table 4.5, we show the p-values for these tests. P-values less than 0.05 imply that the difference in score is statistically significant. These values are highlighted in yellow. Likewise, in Table 4.4, cells corresponding to a statistically significant difference from WhoseFault are also highlighted in yellow. Consider the p-values for all projects combined (see "All Projects" in Table 4.5). At NDCG@1, WhoseFault has performed statistically significantly better than Matter and Tamrawi. These are the only techniques that WhoseFault has performed statistically significantly better than at any level of NDCG.

Table 4.4: Benchmarked Mean NDCG Scores (statistically significant differences from WhoseFault are highlighted)

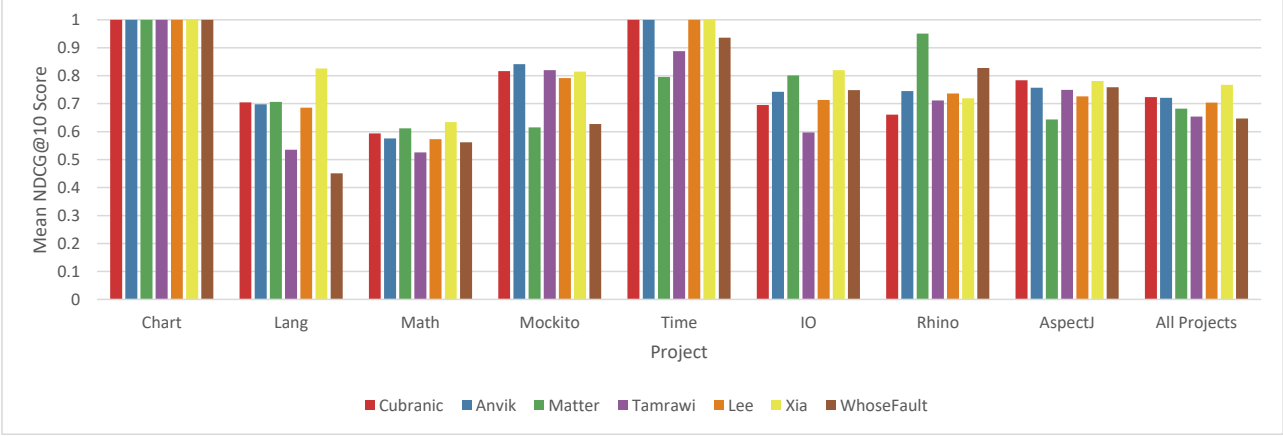
Metric	Project	Cubranic	Anvik	Matter	Tamrawi	Lee	Xia	WhoseFault
NDCG@1	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.556	0.556	0.460	0.254	0.556	0.619	0.286
	Math	0.437	0.417	0.456	0.282	0.388	0.456	0.485
	Mockito	0.763	0.789	0.289	0.763	0.658	0.737	0.500
	Time	1.000	1.000	0.565	0.696	1.000	1.000	0.826
	IO	0.368	0.474	0.526	0.158	0.421	0.579	0.579
	Rhino	0.200	0.333	0.867	0.267	0.333	0.333	0.533
	AspectJ	0.611	0.486	0.333	0.500	0.486	0.556	0.653
	All Projects	0.568	0.550	0.453	0.412	0.524	0.588	0.526
NDCG@5	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.691	0.682	0.682	0.518	0.663	0.823	0.448
	Math	0.572	0.545	0.594	0.491	0.538	0.606	0.556
	Mockito	0.816	0.831	0.534	0.795	0.781	0.797	0.627
	Time	1.000	1.000	0.752	0.888	1.000	1.000	0.936
	IO	0.696	0.742	0.801	0.597	0.676	0.820	0.731
	Rhino	0.661	0.745	0.951	0.712	0.736	0.719	0.828
	AspectJ	0.752	0.738	0.644	0.722	0.684	0.772	0.759
	All Projects	0.708	0.703	0.660	0.632	0.677	0.754	0.644
NDCG@10	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.705	0.698	0.707	0.535	0.686	0.826	0.451
	Math	0.594	0.576	0.612	0.526	0.573	0.634	0.562
	Mockito	0.816	0.841	0.615	0.820	0.792	0.815	0.627
	Time	1.000	1.000	0.796	0.888	1.000	1.000	0.936
	IO	0.696	0.742	0.801	0.597	0.713	0.820	0.749
	Rhino	0.661	0.745	0.951	0.712	0.736	0.719	0.828
	AspectJ	0.784	0.758	0.644	0.750	0.726	0.781	0.759
	All Projects	0.724	0.721	0.683	0.654	0.703	0.768	0.647



(a) Mean NDCG@1 Scores

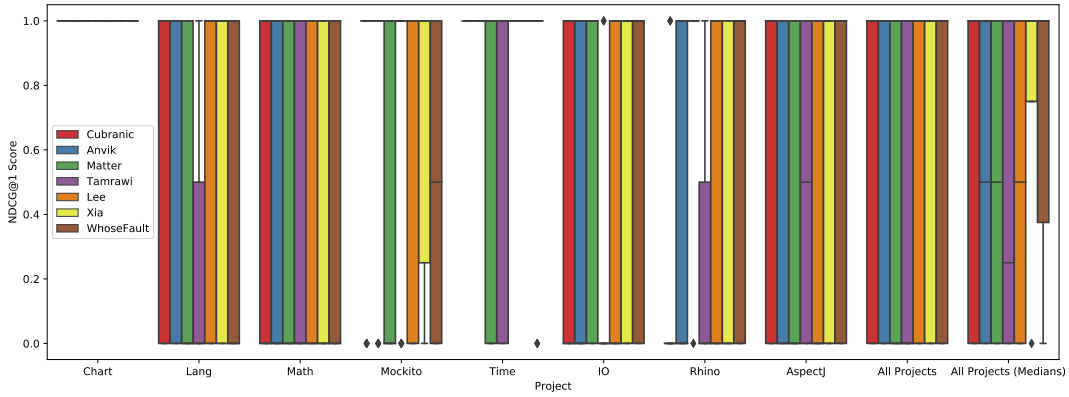


(b) Mean NDCG@5 Scores

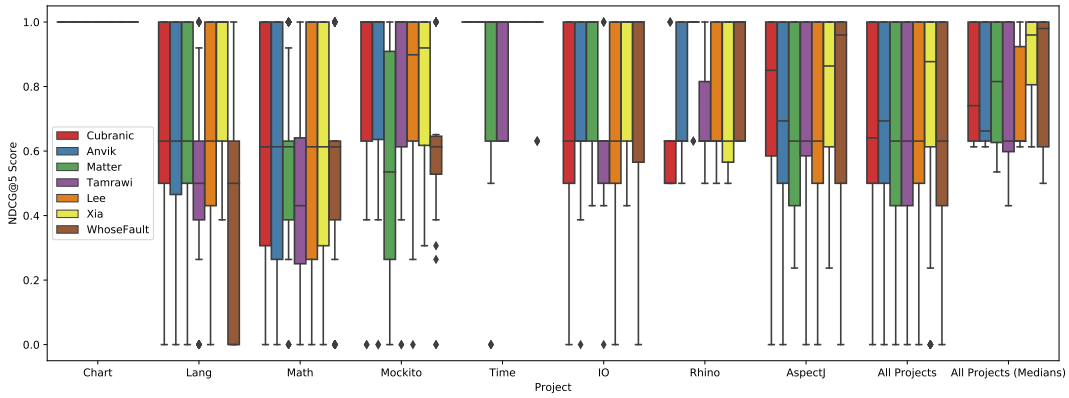


(c) Mean NDCG@10 Scores

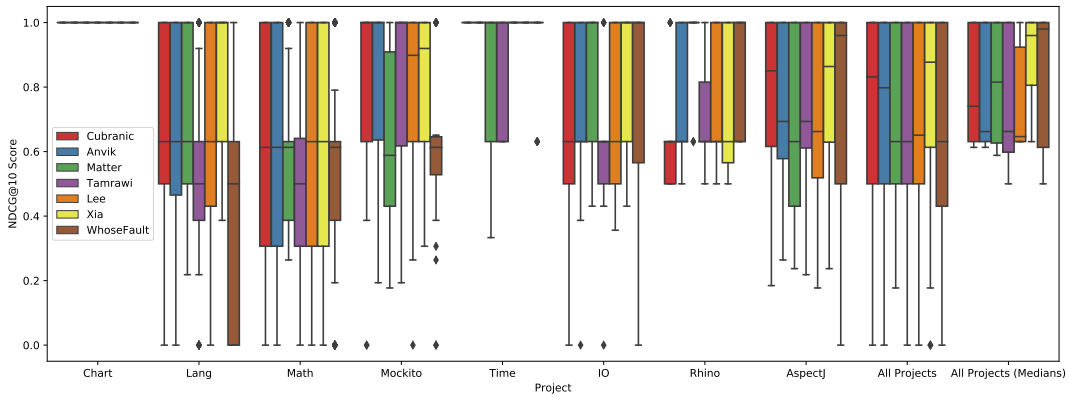
Figure 4.2: Benchmarked Mean NDCG Scores in Bar Charts



(a) NDCG@1 Scores



(b) NDCG@5 Scores



(c) NDCG@10 Scores

Figure 4.3: Benchmarked NDCG Scores in Box Plots

Table 4.5: Wilcoxon Significance Test Between WhoseFault and Report-Based Techniques

Metric	Project	Cubranic	Anvik	Matter	Tamrawi	Lee	Xia
NDCG@1	Chart	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.00	0.00	0.03	0.59	0.00	0.00
	Math	0.47	0.34	0.67	0.00	0.14	0.67
	Mockito	0.03	0.01	0.06	0.02	0.16	0.05
	Time	0.05	0.05	0.03	0.32	0.05	0.05
	IO	0.10	0.32	0.65	0.01	0.26	1.00
	Rhino	0.03	0.08	0.10	0.05	0.18	0.08
	AspectJ	0.47	0.02	0.00	0.03	0.01	0.16
	All Projects	0.21	0.49	0.05	0.00	0.93	0.07
NDCG@5	Chart	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.00	0.00	0.00	0.10	0.00	0.00
	Math	0.47	0.73	0.36	0.09	0.71	0.10
	Mockito	0.00	0.00	0.21	0.00	0.00	0.00
	Time	0.05	0.05	0.01	0.32	0.05	0.05
	IO	0.54	0.83	0.30	0.11	0.53	0.20
	Rhino	0.00	0.06	0.10	0.02	0.11	0.02
	AspectJ	0.73	0.50	0.01	0.15	0.00	0.88
	All Projects	0.00	0.00	0.48	0.69	0.09	0.00
NDCG@10	Chart	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.00	0.00	0.00	0.05	0.00	0.00
	Math	0.28	0.83	0.26	0.31	0.99	0.03
	Mockito	0.00	0.00	0.82	0.00	0.00	0.00
	Time	0.05	0.05	0.02	0.32	0.05	0.05
	IO	0.47	0.94	0.44	0.05	0.58	0.33
	Rhino	0.00	0.06	0.10	0.02	0.11	0.02
	AspectJ	0.40	0.95	0.01	0.59	0.28	0.54
	All Projects	0.00	0.00	0.18	0.60	0.00	0.00

Table 4.6: Average Execution Time (s) Per Bug for Benchmarked Techniques

	Cubranic	**Anvik**	**Matter**	**Tamrawi**	**Lee**	**Xia**	**WhoseFault**
Chart	1.05	0.24	61.89	1.93	55.99	1440.3	35.77
Lang	0.13	0.09	6.44	0.49	41.39	153.88	15.56
Math	0.22	0.17	12.84	0.6	67.99	169.81	111.43
Mockito	0.22	0.14	17.09	0.52	37.19	206.47	23.26
Time	0.12	0.05	9.33	0.29	23.22	65.94	9.26
IO	0.04	0.03	1.93	0.15	20.21	12.8	15
Rhino	0.2	0.19	4.67	0.58	28.15	130.44	2205.83
AspectJ	2.39	3.64	14.48	4.67	92.31	3749.11	448.71
All Projects	0.66	0.87	12.28	1.41	57.07	937.54	233.78

4.2.2 Efficiency

To measure the efficiency of the techniques, we measure the execution time, in seconds, of each technique on each bug in our dataset. To maximize the accuracy of these time measurements, we ran all techniques on the same computer, with no other programs running in the background. Due to time limitations, we only conducted time measurements on a random sample of AspectJ bugs. More specifically, we ran Xia on multiple AspectJ bugs in parallel for effectiveness measurements, and then, we ran, in series, a random sample of 10 AspectJ bugs for time measurement purposes. Other than Xia-AspectJ, we have produced complete time measurements for *all* other technique-project combinations. The results are shown in Figure 4.4 and Table 4.6.

As shown in Table 4.6, Cubranic, Anvik, Matter, Tamrawi, and Lee were more efficient than WhoseFault by an order of magnitude. Nevertheless, WhoseFault took an average of roughly 234 seconds per bug, which is reasonable. Xia, the most effective technique, is the least efficient technique, at roughly 938 seconds per bug.

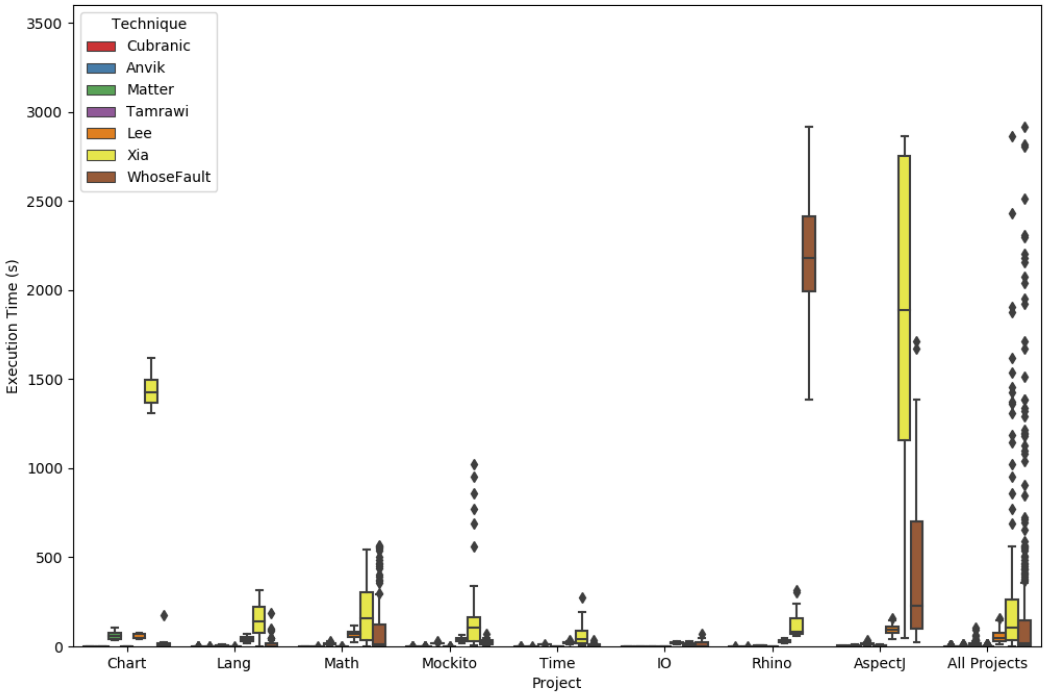


Figure 4.4: Benchmarked Execution Times

4.2.3 Discussion

In this benchmarking experiment, we have compared the performance, in terms of effectiveness and efficiency, of WhoseFault, an execution-based bug assignment technique, and six report-based techniques. Recall that at the beginning of this chapter, we have outlined three possible cases that could be supported by the results of our benchmarking experiment. Our results support *case 1*. That is, our results indicate that *execution-based techniques are worse than report-based techniques*.

First, consider the effectiveness of WhoseFault compared to the report-based techniques (see Figure 4.2). At NDCG@1, WhoseFault has performed statistically significantly better than Matter and Tamrawi. WhoseFault also performed better than Lee, at NDCG@1, but not by a statistically significantly difference. Although WhoseFault performed worse than Cubranic, Anvik, and Xia at NDCG@1, our significance test did not indicate that these differences were statistically significantly better. At NDCG@5, WhoseFault is less effective than all the other techniques except for Tamrawi. Cubranic, Anvik, and Xia have all performed statistically significantly better than WhoseFault, at NDCG@10.

Next, consider the efficiency of WhoseFault compared to the report-based techniques (see Table 4.6). WhoseFault is slower than all the other techniques, expect for Xia, by an order of magnitude. However, it is roughly four times faster than Xia, the most effective technique in our experiment. Even though WhoseFault is slower than most of the benchmarked techniques, its efficiency is still reasonable - on average, WhoseFault took 234 seconds to execute on a single bug. Therefore, there is not a big difference between WhoseFault and the other techniques in terms of efficiency.

These observations indicate the following conclusions. Report-based techniques are better, but execution-based techniques are not much worse. If both a bug report as well as an

execution are available, one should probably use either Xia for effectiveness or Cubranic for efficiency. If only an execution, and not a report, is available, then one should consider the option of writing a bug report. If a high-quality report could be efficiently written, then one could use a report-based technique. Otherwise, an execution-based technique should be used. The fact that the execution-based techniques are not much worse than report-based techniques should provide confidence in the scenarios in which only execution-based techniques are applicable.

Chapter 5

DR_BEV: Developer Recommendation Based on Executed Vocabulary

The objective of an execution-based expertise finding technique is to rank developers in descending order of expertise towards a given execution-based bug. To the best of our knowledge, the only existing execution-based expertise finding technique is WhoseFault [15]. Therefore, we consider WhoseFault to be the state-of-the-art execution-based expertise finding technique. WhoseFault models a bug using Tarantula, a fault localization technique. The idea behind WhoseFault is that developers who have recently modified highly suspicious lines of code are most likely to have the necessary expertise to fix a bug. At the core of WhoseFault is a heuristic formula that assigns each developer an expertise score based on the recency and fault localization suspiciousness of the developer's line-level code modifications.

In this chapter, we propose an alternative execution-based bug assignment technique, called *DR_BEV* (***D***eveloper ***R***ecommendation ***B***ased on ***E***xecuted ***V***ocabulary), that models expertise with a vocabulary-based expertise model, as opposed to a location-based expertise model. The idea behind *DR_BEV* is that a developer demonstrates expertise by the vocabulary in their source code contributions, and this expertise can be applied to anywhere in the code where that vocabulary exists. *DR_BEV* is inspired by the work of Matter et al. [52].

First, we present the idea of *DR_BEV*. Then, we provide details of our implementation, and lastly, we test *DR_BEV* on the dataset described in Chapter 4.1.2.

5.1 Approach

We model *DR_BEV* as an information retrieval system in which documents represent developers and a search query represents a bug. Given a bug, our objective is to rank the developers in descending order of expertise towards the bug. To model the expertise of each developer as a document, we use their source code contributions. To model the bug as a search query, we use the buggy code execution coverage source code as text.

Let $\mathcal{D} = \{d_1, \ldots, d_m\}$ be the set of m developers. Let $V = \{w_1, \ldots, w_n\}$ be the vocabulary of n terms that occur in the code repository. Let M_{mxn} be a term-document matrix (TDM) with m rows and n columns. The value $M_{i,j}$ is equal to the number of times that the term w_j occurs in the source code contributions of developer d_i. Developer d_i's expertise is modeled by M_i, the ith row of M. Let Q be a query vector, of length n, that represents a bug. The value of Q_j is equal to the number of times that term w_j occurs in the code coverage of the buggy execution. To rank the m developers by their expertise towards the bug, *DR_BEV* proceeds as follows. For each developer d_i, *DR_BEV* computes the cosine similarity between M_i and Q as the developer's expertise score. It then ranks the developers by descending expertise scores.

As shown in Figure 5.1, *DR_BEV* has three main stages: expertise modeling, bug modeling, and developer ranking. In the expertise modeling stage, we model developer expertise in a TDM based on developers' source code contributions in the project's code repository. Next, in the bug modeling stage, we model the bug as a query vector based on the buggy code execution. Lastly, in the developer ranking stage, we score developer expertise based on

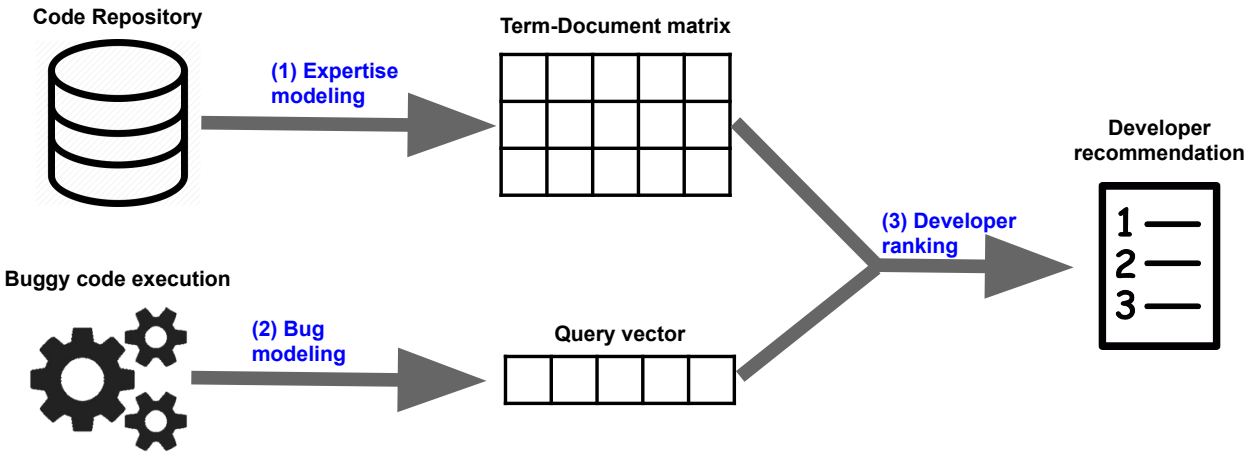


Figure 5.1: *DR_BEV* Diagram

cosine similarity, and rank developers by descending expertise scores.

We also perform two time-based expertise penalty techniques introduced by Matter et al. [52]:

- *Vocabulary decay*: For each commit, the word counts are weighted by a decay factor that is a factor of the age of the commit. We calculate the time difference, in weeks, between today and the date of the commit. The word counts are decayed by a factor of 3% for each week.

- *Inactive developer penalty*: For each developer, we calculate the time difference, in years, between today and the date of the developer's latest commit. Then, we penalize each developer's expertise score by 0.2 for each year in this calculated time difference.

5.1.1 Expertise Modeling

Developer expertise is modeled from the developers' source code contributions in the project's code repository. This stage has three steps: (1) code repository mining, (2) text processing, and (3) vectorization.

In the code repository mining step, we mine each commit in the project's code repository as follows. We retrieve the identification of the author who made the commit, the timestamp of when the commit was made, and the commit's source code contribution, which, following Matter et al.'s [52] definition, we define as comprising of the following components:

- *Added code*: When a developer adds line of source code, they show expertise for those lines because it is required to have expertise in the vocabulary of the added code.
- *Deleted code*: Likewise, it is also required to have expertise in the vocabulary of deleted code.
- *Context code*: The vocabulary of code near the added or deleted code is relevant to those added or deleted lines. We used git's default three lines of context above and below blocks of added or deleted code.
- *Commit message*: Each commit has a short textual message that briefly describes the changes made.

In the text processing step, we process the text of each source code contribution, which contains the components indicated above. We tokenize by camel-case and non-alphabetic characters. In order to reduce the size and complexity of the data, we perform stemming and stop word removal. We stem each token to its root form. For example, *connect*, *connected*, and *connecting* are all stemmed to the root *connect*. We remove stop words such as *a*, *but*, and *to*.

Finally, we vectorize the processed text. For each developer, we proceed as follows. We create an empty bag-of-words vector of length n. For each commit made by the developer, we take the processed text, and count word occurrences to create a vector v of word counts. Then, we decay v in accordance to the *vocabulary decay* technique, and lastly, we add this decayed

vector of counts to the developer's bag-of-words vector. Each developer's bag-of-words vector represents their expertise.

5.1.2 Bug Modeling

The bug is modeled from a buggy execution of code. The lines of source code that were executed by a failing test case are used as the query text. To create the query vector Q, this query text is vectorized in the same fashion described in the text processing step of the expertise modeling stage.

5.1.3 Developer Ranking

Finally, in the developer ranking stage, we rank the developers by descending score of expertise towards the bug. For developer d_i, the expertise vector M_i is the i-th row of M. The bug is represented by the query vector Q. To compare the bug vector to the developer vectors, we use cosine similarity, as shown in Equation 5.1. The expertise score for developer d_i is the cosine similarity between M_i and Q. Lastly, we rank the developers by descending expertise scores.

$$\text{Similarity}(\boldsymbol{A}, \boldsymbol{B}) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{||\boldsymbol{A}||\,||\boldsymbol{B}||} \tag{5.1}$$

5.2 Experimental Implementation

We implemented our technique in Python. We used PyDriller [71] to mine Git repositories. For stemming, we used the Porter Stemmer [77], and for stop word removal, we used

NLTK's [70] list of stop words.

5.3 Research Method

We evaluate *DR_BEV* on the bug dataset described in Chapter 4.1.2. We investigate *DR_BEV*'s effectiveness and efficiency in comparison to WhoseFault, the state-of-the-art execution-based technique, as well as to report-based techniques (see Chapter 4.1.1). We address effectiveness in order to give the user confidence in using *DR_BEV*, and we address efficiency in order to determine *DR_BEV*'s practicality. We measure effectiveness using NDCG@1, NDCG@5, NDCG@10, and we measure effiency by measuring the execution time of each technique on each bug.

In this chapter, we seek to answer the following research questions:

4. How does *DR_BEV* compare to other techniques in terms of *effectiveness*?

5. How does *DR_BEV* compare to other techniques in terms of *efficiency*?

6. How do various parameter settings affect *DR_BEV*'s effectiveness?

5.4 Results

5.4.1 Effectiveness

We compare the effectiveness of *DR_BEV* to that of the benchmarked techniques (see Chapter 4.1.1), and we perform Wilcoxon significance tests between *DR_BEV*'s scores and other techniques' scores. We report mean effectiveness scores in Table 5.1. We show the

effectiveness comparison in Figure 5.2 as bar charts and in Figure 5.3 as box plots. In both figures, we report scores grouped by project, as well as all projects combined in the *All Projects* group. In Table 5.2, we report Wilcoxon p-values. In both Table 5.2 and Table 5.2, we highlight values, which are less than or equal to 0.05, that indicate a statistically significant difference in scores.

At all three levels of NDCG, the most effective technique was Xia.

Consider the effectiveness of *DR_BEV* in comparison to that of the six report-based techniques (i.e. Cubranic, Anvik, Matter, Tamrawi, Lee, and Xia). At NDCG@1, NDCG@5, and NDCG@10, *DR_BEV* was more effective than three, two, and two of the six report-based techniques, respectively, in the *All Projects* group.

Next, consider the effectiveness of *DR_BEV* in comparison to that of WhoseFault, the only existing execution-based technique. At all three levels of NDCG, *DR_BEV* was more effective than WhoseFault in the *All Projects* group.

Finally, consider the Wilcoxon p-values shown in Table 5.2. At NDCG@1, no technique has performed statistically significantly better than *DR_BEV*. At NDCG@5, only one technique, namely Xia, performed statistically significantly better than *DR_BEV*. At NDCG@10, two techniques, namely Anvik and Xia, performed statistically significantly better than *DR_BEV*.

5.4.2 Efficiency

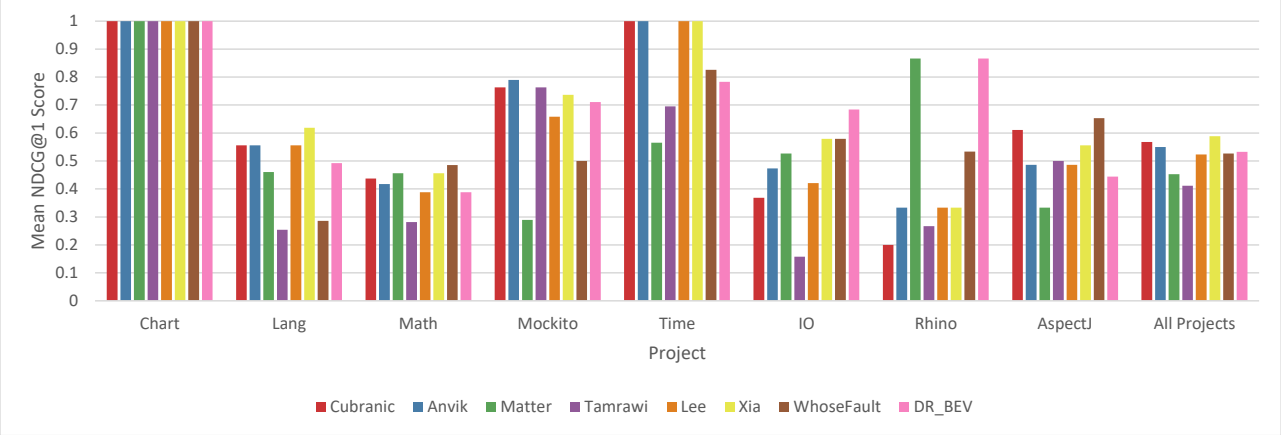
Table 5.3 shows the mean execution time, in seconds, of each technique on each project's bugs, as well as on *All Projects* combined. Consider the efficiency of each technique in the *All Projects* group. *DR_BEV* was faster than three techniques (i.e. Lee, Xia, WhoseFault), and slower than four (i.e. Cubranic, Avik, Matter, Tamrawi). On the average bug, *DR_BEV*

Table 5.1: Mean NDCG Scores for *DR_BEV* and Existing Techniques (statistically significant differences from *DR_BEV* are highlighted)

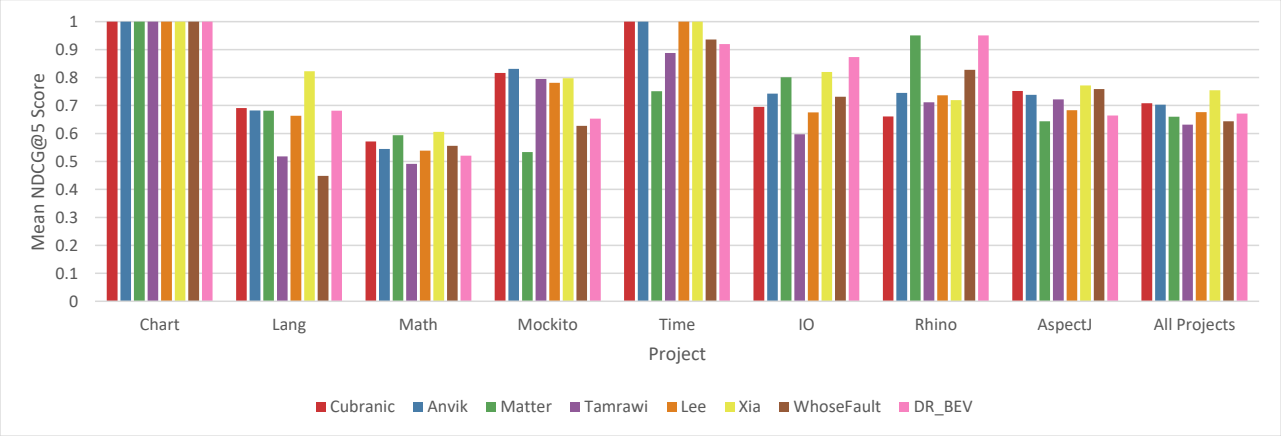
Metric	Project	Cubranic	Anvik	Matter	Tamrawi	Lee	Xia	WhoseFault	*DR_BEV*
NDCG@1	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.556	0.556	0.460	0.254	0.556	0.619	0.286	0.492
	Math	0.437	0.417	0.456	0.282	0.388	0.456	0.485	0.388
	Mockito	0.763	0.789	0.289	0.763	0.658	0.737	0.500	0.711
	Time	1.000	1.000	0.565	0.696	1.000	1.000	0.826	0.783
	IO	0.368	0.474	0.526	0.158	0.421	0.579	0.579	0.684
	Rhino	0.200	0.333	0.867	0.267	0.333	0.333	0.533	0.867
	AspectJ	0.611	0.486	0.333	0.500	0.486	0.556	0.653	0.444
	All Projects	0.568	0.550	0.453	0.412	0.524	0.588	0.526	0.532
NDCG@5	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.691	0.682	0.682	0.518	0.663	0.823	0.448	0.682
	Math	0.572	0.545	0.594	0.491	0.538	0.606	0.556	0.521
	Mockito	0.816	0.831	0.534	0.795	0.781	0.797	0.627	0.653
	Time	1.000	1.000	0.752	0.888	1.000	1.000	0.936	0.920
	IO	0.696	0.742	0.801	0.597	0.676	0.820	0.731	0.873
	Rhino	0.661	0.745	0.951	0.712	0.736	0.719	0.828	0.951
	AspectJ	0.752	0.738	0.644	0.722	0.684	0.772	0.759	0.664
	All Projects	0.708	0.703	0.660	0.632	0.677	0.754	0.644	0.671
NDCG@10	Chart	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Lang	0.705	0.698	0.707	0.535	0.686	0.826	0.451	0.707
	Math	0.594	0.576	0.612	0.526	0.573	0.634	0.562	0.565
	Mockito	0.816	0.841	0.615	0.820	0.792	0.815	0.627	0.677
	Time	1.000	1.000	0.796	0.888	1.000	1.000	0.936	0.920
	IO	0.696	0.742	0.801	0.597	0.713	0.820	0.749	0.873
	Rhino	0.661	0.745	0.951	0.712	0.736	0.719	0.828	0.951
	AspectJ	0.784	0.758	0.644	0.750	0.726	0.781	0.759	0.664
	All Projects	0.724	0.721	0.683	0.654	0.703	0.768	0.647	0.692

Table 5.2: Wilcoxon Significance Test Between *DR_BEV* and Existing Techniques

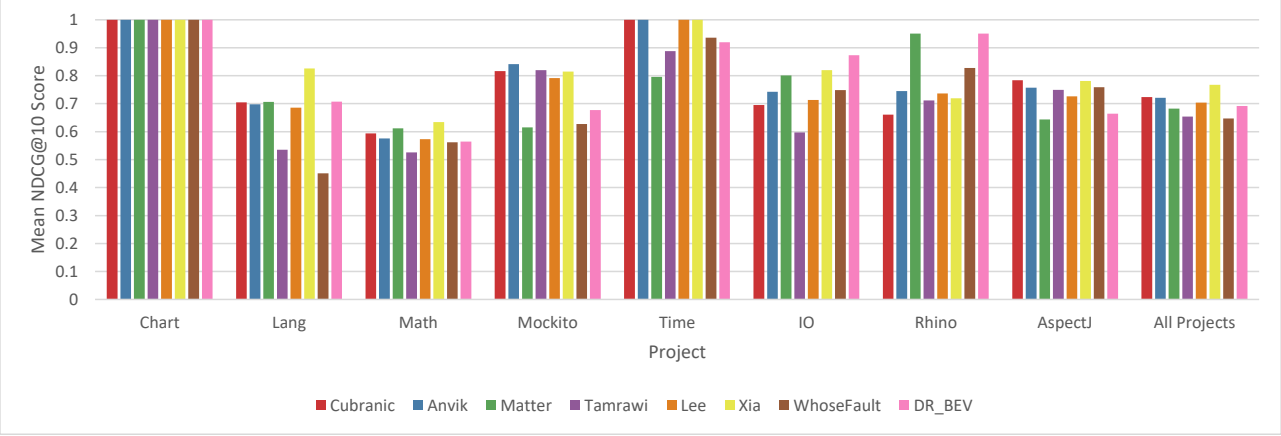
Metric	Project	Cubranic	Anvik	Matter	Tamrawi	Lee	Xia	WhoseFault
NDCG@1	Chart	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.41	0.43	0.67	0.01	0.41	0.09	0.02
	Math	0.48	0.67	0.26	0.11	1.00	0.33	0.08
	Mockito	0.48	0.26	0.00	0.56	0.53	0.76	0.05
	Time	0.03	0.03	0.10	0.41	0.03	0.03	0.74
	IO	0.06	0.16	0.18	0.00	0.13	0.53	0.48
	Rhino	0.00	0.02	1.00	0.01	0.00	0.00	0.10
	AspectJ	0.01	0.58	0.19	0.41	0.55	0.09	0.01
	All Projects	0.29	0.61	0.02	0.00	0.79	0.09	0.86
NDCG@5	Chart	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.94	0.90	0.81	0.00	0.68	0.00	0.00
	Math	0.30	0.73	0.02	0.28	0.81	0.05	0.21
	Mockito	0.00	0.00	0.07	0.00	0.00	0.00	0.28
	Time	0.03	0.03	0.05	0.41	0.03	0.03	0.74
	IO	0.02	0.08	0.13	0.00	0.05	0.43	0.13
	Rhino	0.00	0.02	1.00	0.01	0.01	0.01	0.10
	AspectJ	0.01	0.01	0.52	0.08	0.58	0.00	0.01
	All Projects	0.08	0.11	0.62	0.04	0.80	0.00	0.25
NDCG@10	Chart	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Lang	0.98	0.91	0.85	0.00	0.84	0.00	0.00
	Math	0.48	0.76	0.09	0.18	0.81	0.07	0.80
	Mockito	0.00	0.00	0.29	0.00	0.01	0.00	0.07
	Time	0.03	0.03	0.05	0.41	0.03	0.03	0.74
	IO	0.02	0.08	0.13	0.00	0.08	0.43	0.13
	Rhino	0.00	0.02	1.00	0.01	0.01	0.01	0.10
	AspectJ	0.00	0.00	0.52	0.01	0.04	0.00	0.01
	All Projects	0.06	0.05	0.50	0.04	0.34	0.00	0.04



(a) Mean NDCG@1 Scores

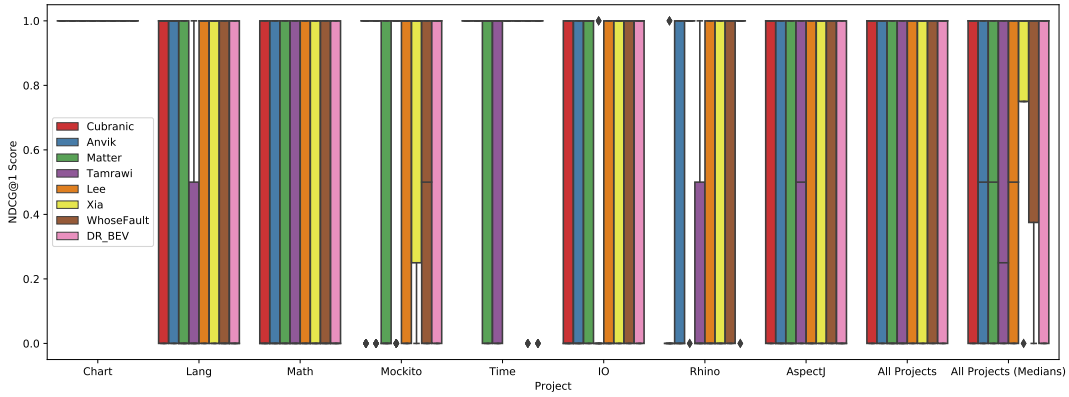


(b) Mean NDCG@5 Scores

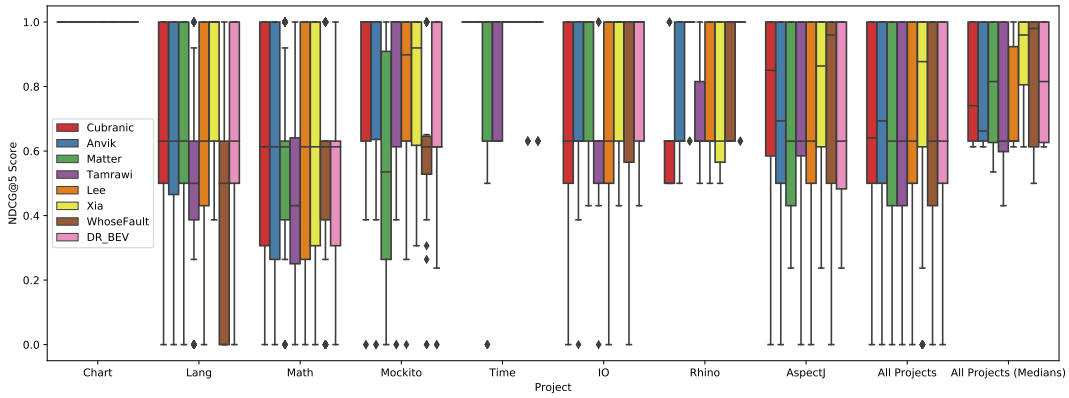


(c) Mean NDCG@10 Scores

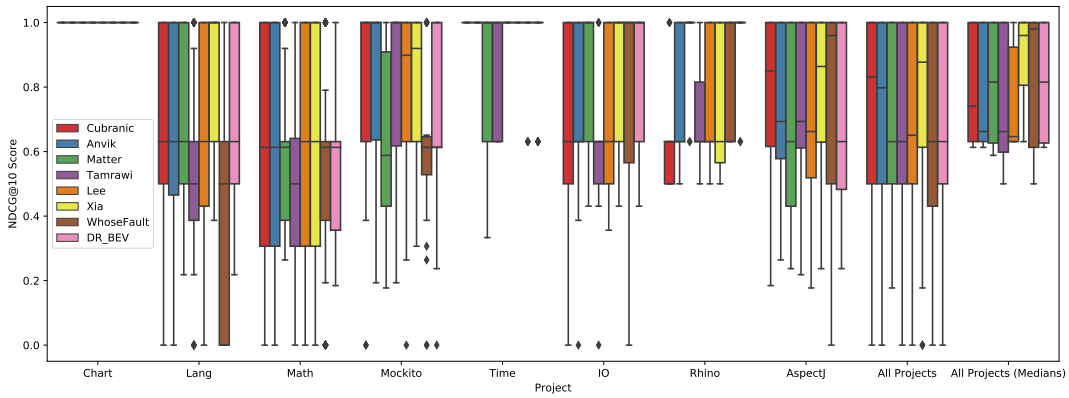
Figure 5.2: Effectiveness of *DR_BEV* and Other Techniques in Bar Charts



(a) NDCG@1 Scores



(b) NDCG@5 Scores



(c) NDCG@10 Scores

Figure 5.3: Effectiveness of *DR_BEV* and Other Techniques in Box Plots

Table 5.3: Mean Execution Time (s) Per Bug for *DR_BEV* and Existing Techniques

	Cubranic	**Anvik**	**Matter**	**Tamrawi**	**Lee**	**Xia**	**WhoseFault**	***DR_BEV***
Chart	1.05	0.24	61.89	1.93	55.99	1440.30	35.77	74.21
Lang	0.13	0.09	6.44	0.49	41.39	153.88	15.56	7.17
Math	0.22	0.17	12.84	0.60	67.99	169.81	111.43	13.45
Mockito	0.22	0.14	17.09	0.52	37.19	206.47	23.26	14.03
Time	0.12	0.05	9.33	0.29	23.22	65.94	9.26	9.80
IO	0.04	0.03	1.93	0.15	20.21	12.80	15.00	2.38
Rhino	0.20	0.19	4.67	0.58	28.15	130.44	2205.83	6.13
AspectJ	2.39	3.64	14.48	4.67	92.31	3749.11	448.71	20.24
All Projects	0.66	0.87	12.28	1.41	57.07	937.54	233.78	13.85

was roughly 68 times faster than Xia, which was the most effective technique (see Figure 5.2).

5.4.3 Sensitivity Analyses

We performed the following sensitivity analyses on *DR_BEV*:

- *Use of Time-Based Expertise Penalties*: We compare the effectiveness of using *vocabulary decay* only, *inactive developer penalty* only, both penalties, and neither penalty.

- *Percentage of Code Coverage Used*: Each line of source code that has a fault localization suspiciousness score greater than 0.0 was executed, or *covered*, by a failed test case. The buggy code coverage consists of all lines of source code with a suspiciousness score greater than 0.0. We test our technique where the buggy source code consists of different percentages of the most suspiciousness lines of source code. More specifically, we test the use of 5%, 10%, 20%, 30%, ..., 100% of the most suspiciousness source code lines.

- *Percentage of Project History Used*: We compare the effectiveness of using the whole code repository history for expertise modeling to the effectiveness of using some percentage of the most recent history in the code repository. More specifically, we test the use 5%, 10%, 20%, 30%, ..., 100% of the project's most recent history.

- *Text Vectorization Methods*: We compare the effectiveness of count vectorization and TF-IDF vectorization.

Figure 5.4 shows the results of the *Use of Time-Based Expertise Penalties* analysis. The results indicate that the inactive developer penalty strongly improves *DR_BEV*'s effectiveness. The two configurations in which the inactive developer penalty was used (i.e. *Inactive Developer* and *Both*) were more effective than the other two configurations (i.e. *None* and *Vocabulary Decay*).

Figure 5.5 shows the results of the *Percentage of Code Coverage Used* analysis. Consider the effectiveness of the *All Projects* group. At NDCG@1, using 100% and 60% of code coverage yielded the highest effectiveness. At NDCG@5 and NDCG@10, 40% and 100%, respectively of code coverage yielded the highest effectiveness. One possible explanation for why using 40% or 60% is more effective than using 100% is that Tarantula, the fault localization technique used, was very accurate, and therefore, using only some percentage of the most suspicious lines generates a better bug model than using the entire buggy code coverage does. However, because the percentage of code coverage used does not strongly affect *DR_BEV*'s effectiveness, one should be confident in using an entire buggy code coverage in the case that fault localization data is unavailable.

Figure 5.6 shows the results of the *Percentage of Project History Used* analysis. In the *All Projects* group, the highest effectiveness at NDCG@1, NDCG@5, and NDCG@10 was achieved when 10%, 5%, and 10%, respectively, of the project's history was used. One

possible explanation for why using only the recent commits, rather than the project's entire commit history, is more effective is that the developer's vocabulary of expertise changes quickly with time. Therefore, only the most recent history is a good representation of the developers' expertise. Another possible explanation is that, often, the developer who is most suitable to fix a bug has very recently made contributions to the source code, and therefore, the most recent history alone is a strong indication of expertise towards the bug.

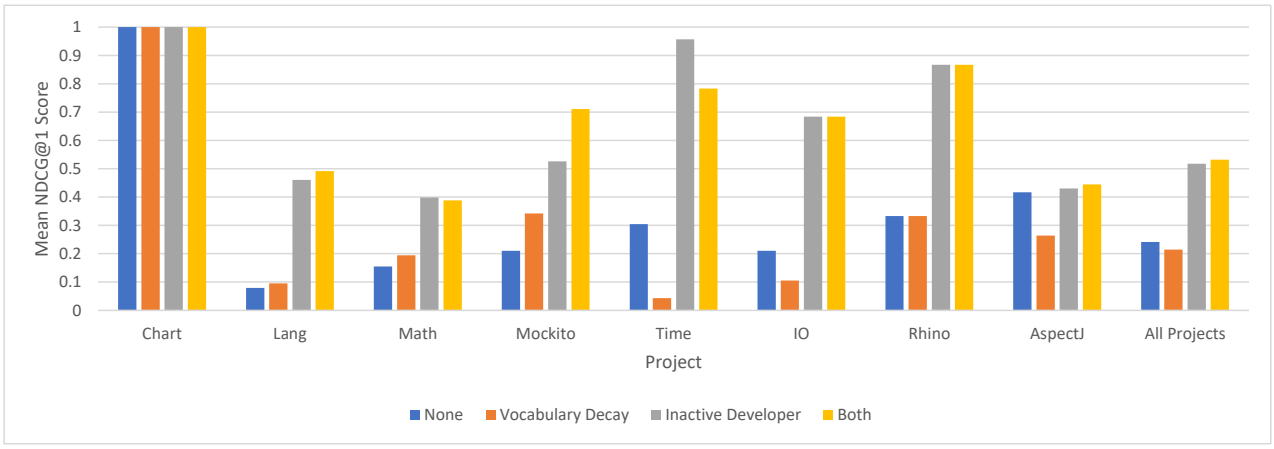
Figure 5.7 shows the results of the *Text Vectorization Methods* analysis. At all three levels of NDCG, TF-IDF vectorization was more effective than count vectorization in the *All Projects* group.

Collectively, the sensitivity analyses indicate that some fine tuning of parameters can improve *DR_BEV*'s effectiveness. The fourth sensitivity analysis (see Figure 5.7) indicates that replacing count vectorization with TF-IDF vectorization improves the mean NDCG@5 score for *All Projects* roughly from 0.67 to 0.71. Additionally, the first sensitivity analysis (see Figure 5.4) suggests that the expertise penalty techniques can be fine tuned.

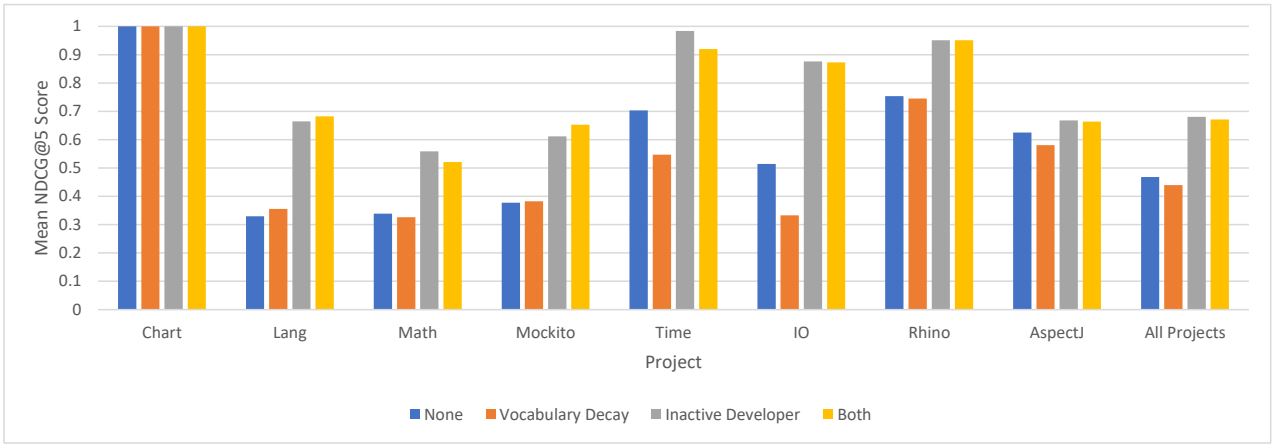
5.4.4 Discussion

Recall that in Chapter 4, our benchmarking experiment supported *case 1: execution-based techniques are worse than report-based techniques*. We observed that while WhoseFault, the current state-of-the-art execution-based technique, is worse than existing report-based techniques, it is not worse by much.

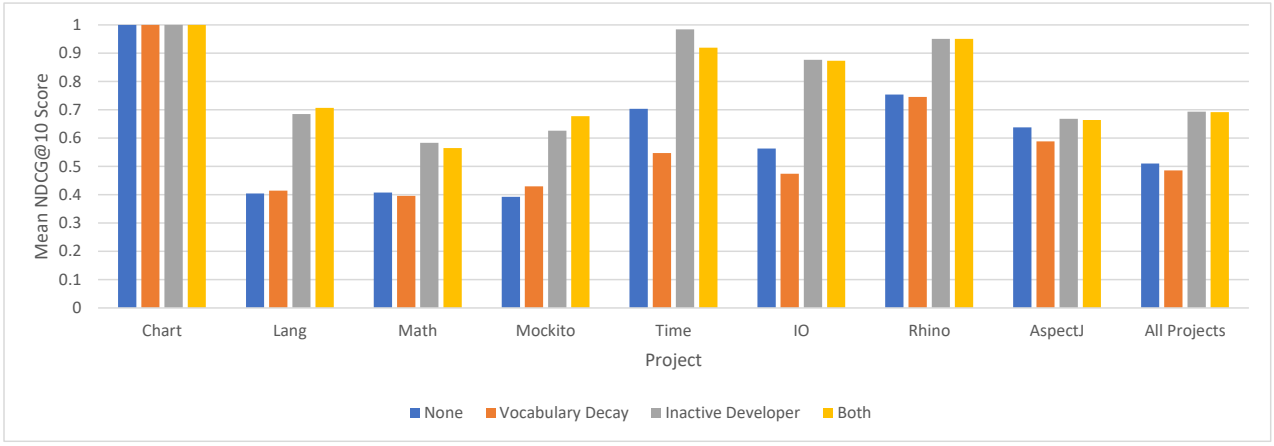
In this chapter, we proposed *DR_BEV*, a novel execution-based bug assignment technique based on the vocabulary in a buggy code execution. The results of the experiment in this current chapter further support the conclusion that execution-based techniques are worse, but not by much, than report-based techniques. At NDCG@5, Xia was the only technique



(a) Mean NDCG@1 Scores

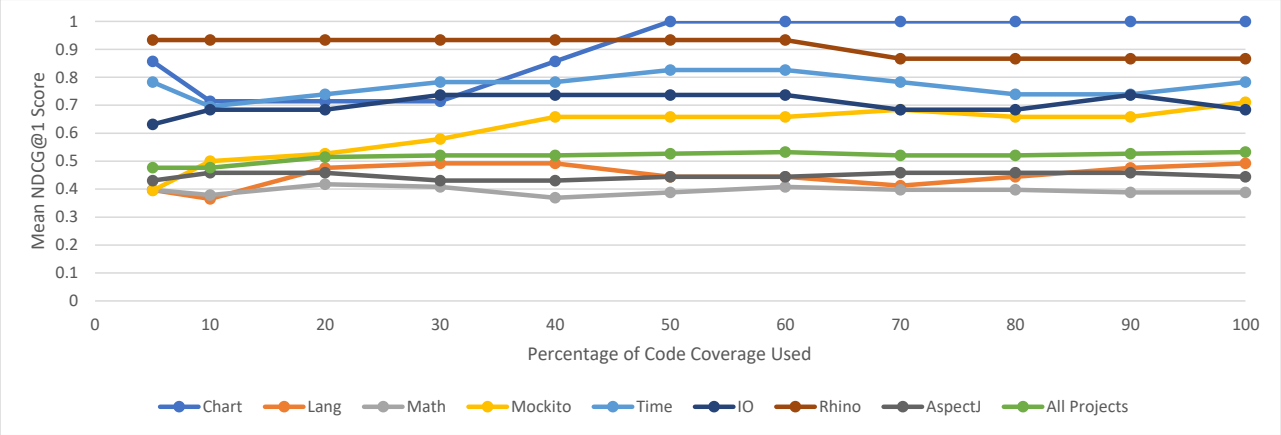


(b) Mean NDCG@5 Scores

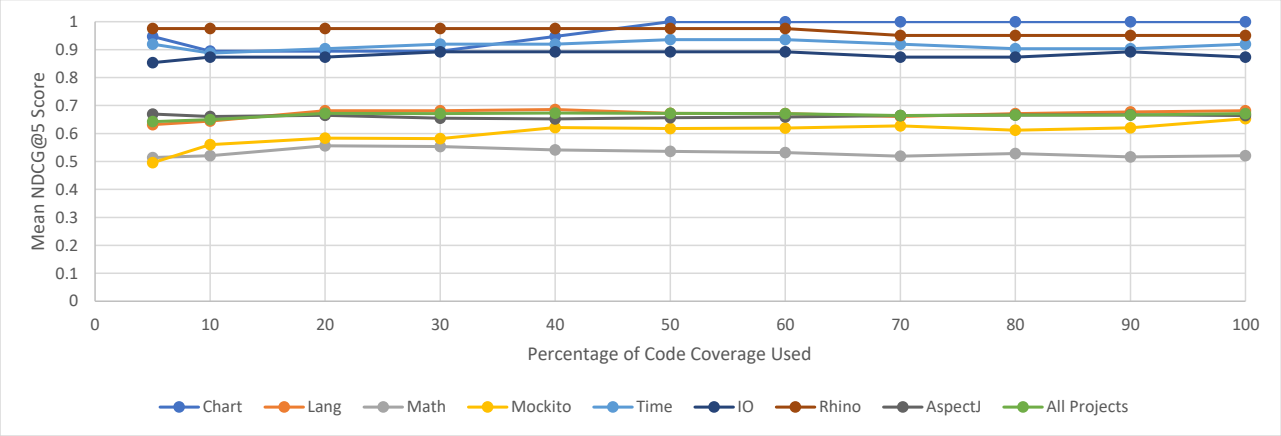


(c) Mean NDCG@10 Scores

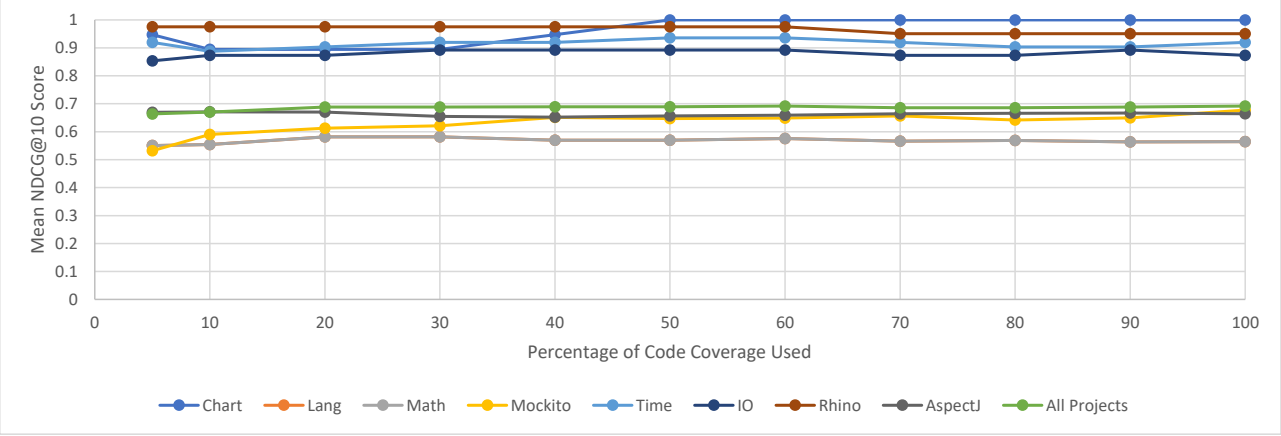
Figure 5.4: Sensitivity Analysis #1: Use of Time-Based Expertise Penalties



(a) Mean NDCG@1 Scores

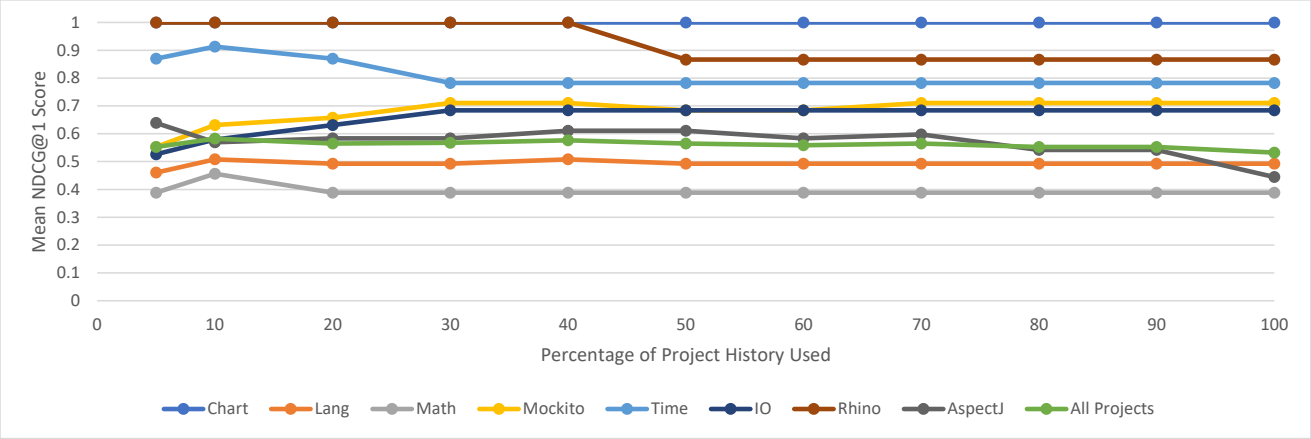


(b) Mean NDCG@5 Scores

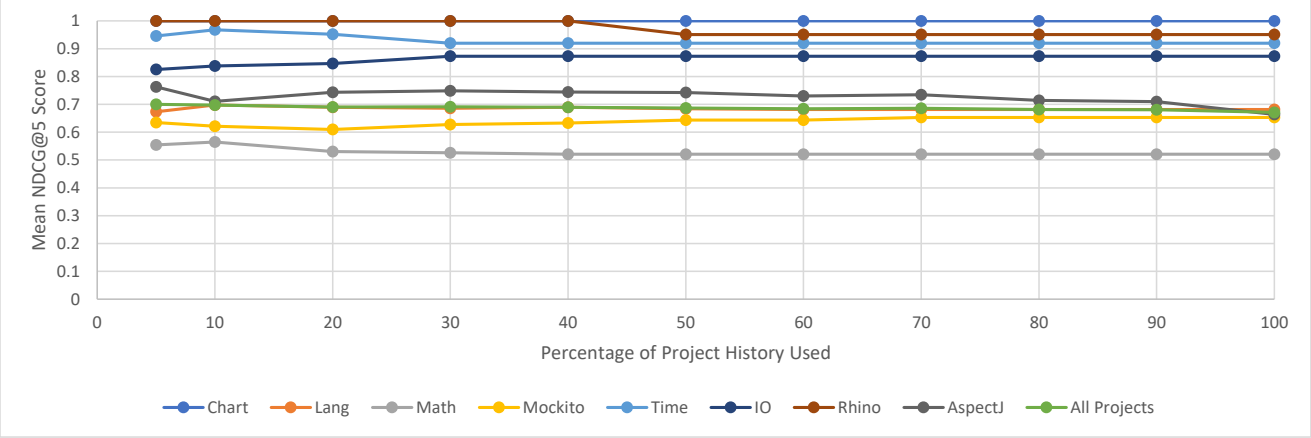


(c) Mean NDCG@10 Scores

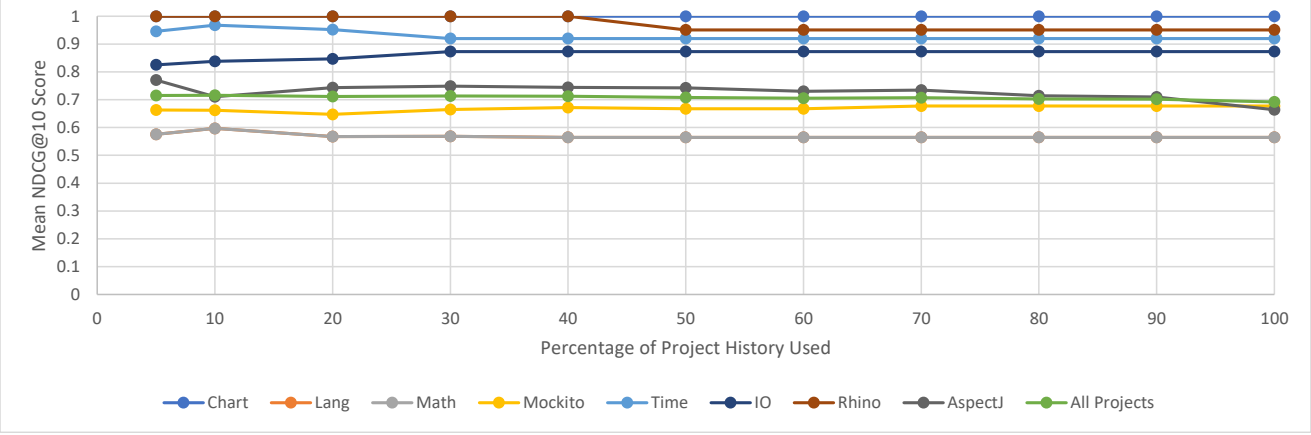
Figure 5.5: Sensitivity Analysis #2: Percentage of Code Coverage Used



(a) Mean NDCG@1 Scores

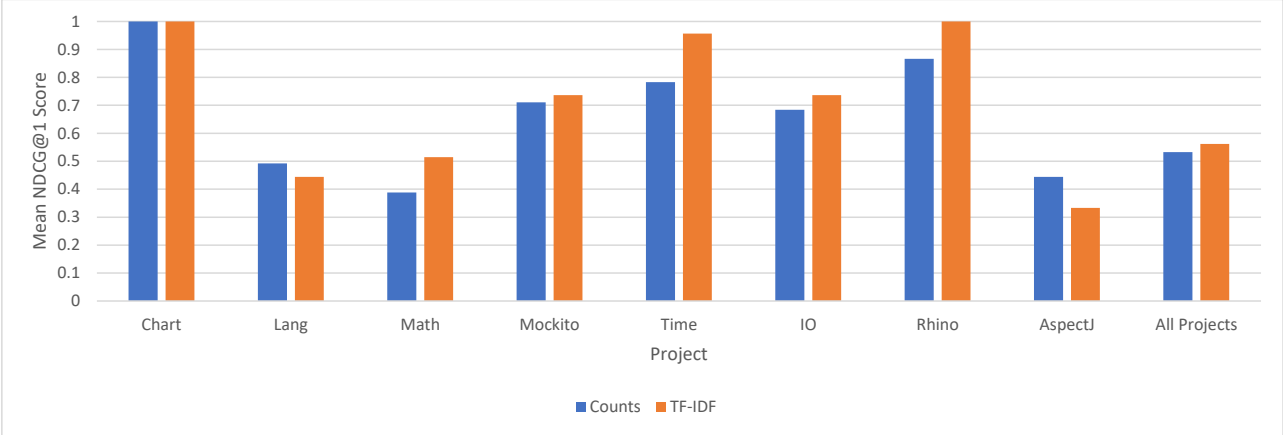


(b) Mean NDCG@5 Scores

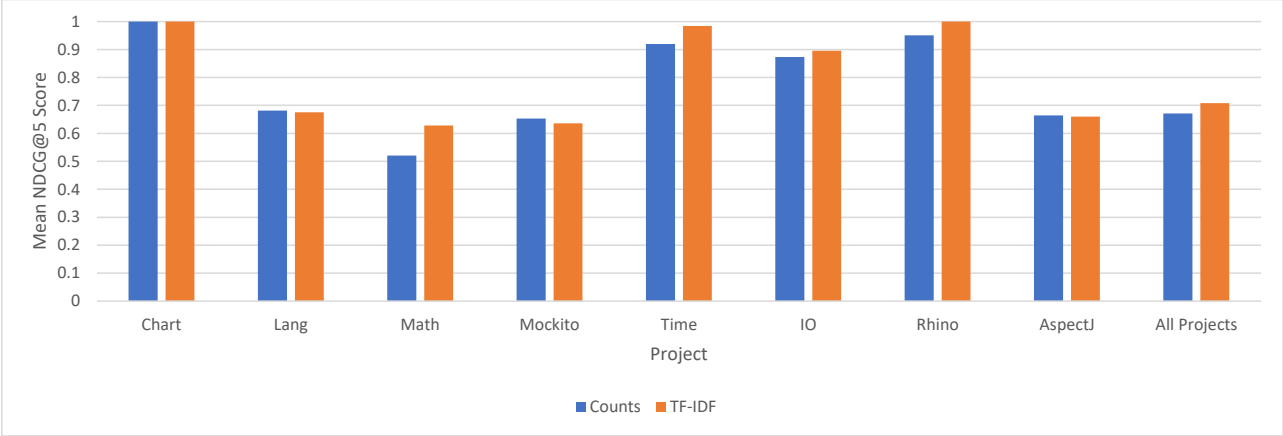


(c) Mean NDCG@10 Scores

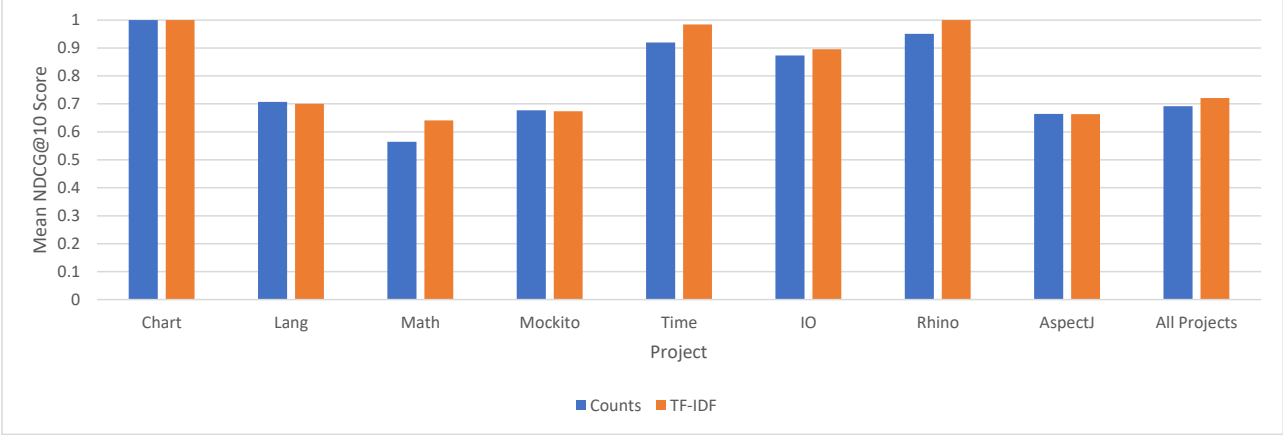
Figure 5.6: Sensitivity Analysis #3: Percentage of Project History Used



(a) Mean NDCG@1 Scores



(b) Mean NDCG@5 Scores



(c) Mean NDCG@10 Scores

Figure 5.7: Sensitivity Analysis #4: Text Vectorization Methods

that was statistically significantly better than *DR_BEV*, and no technique was statistically significantly better than *DR_BEV* at *all* three levels of NDCG. *DR_BEV* is 68 times more efficient than Xia, the most effective technique. Our experiment indicates that a user should have confidence in *DR_BEV*'s performance. However, if a high-quality bug report can be efficiently written, then one could use Xia for higher effectiveness, albeit lower efficiency.

Chapter 6

Conclusions

In this thesis, we made four main research contributions. First, we summarized existing bug assignment techniques and classified them into a hierarchy of families of similar techniques. To the best of our knowledge, we have included all published techniques as of the time of writing. Second, we conducted an empirical study of bug-fix commits in open-source software projects. In this study, we found out that there exist situations in which a failed execution alone, and *not* a bug report, is available for a bug. Third, we benchmarked existing techniques in order to compare WhoseFault, an execution-based technique, to report-based techniques. To the best of our knowledge, this is the first study to compare execution-based and report-based techniques. Our results in this benchmarking experiment indicate that while report-based techniques perform better than execution-based techniques, execution-based techniques perform competitively with report-based techniques. Lastly, we proposed a novel execution-based technique, *DR_BEV*, which models developer expertise based on the vocabulary of each developer's source code contributions. Our experiment indicates that *DR_BEV* is more effective and more efficient than the existing state-of-the-art execution-based technique.

6.1 Threats to Validity

6.1.1 Threats to Internal Validity

Threats to internal validity relate to possible errors in the experimentation.

One threat to internal validity in our work has to do with our replication of existing bug assignment techniques (see Chapter 4.1.1). We have done our best to replicate each technique as accurately as possible. When we were unsure of any intricate details, we made what we believed to be the most natural choice for the technique's design.

Another threat to internal validity has to do with the mapping of developer identifiers in software repositories (see Chapter 4.1.3). There might have been errors in our mapping of developer identifiers. The identifiers in the code repositories have mostly consisted of a full name followed by an email address. In the bug repositories, the identifiers have mostly consisted of usernames. We have done our best to match identifiers between the bug and code repositories. For example, given a username in the bug repositories, we opened the webpage corresponding to that username in the online bug repository. In some instances, we were able to see the developer's full name on that page. In other instances, however, we were not. In those cases in which we were unsure, we made our best judgement regarding which identifiers match.

A third threat to internal validity has to do with parameter optimization. Our settings for the parameters of each technique might not have been optimal for each software project studied, which could affect our results. However, our objective was to determine how well predetermined parameters, which were presented in each technique's original paper, suit new software projects. In order to determine the results when each technique's parameters are optimized for each software project, one would have to optimize those parameters for each

technique and for each individual software project.

6.1.2 Threats to External Validity

Threats to external validity relate to the generalizability of the experimental results. In our work, we evaluated the studied techniques on eight open-source software projects. Due to the very large variety of software projects, we are unable to definitively state that our findings will hold for any software project in general. However, to reduce this threat, we conducted our study on real software projects that are extensively used in real-world applications. To further minimize this threat, more bugs from other software projects should be analyzed.

6.1.3 Threats to Construct Validity

Threats to construct validity relate to the suitability of the evaluation metric used. In our case, we scored each technique's effectiveness based on the location of the ground truth developer in a ranked list of recommended developers. However, there are a number of possible scenarios in which the ground truth developer is not the one most suitable to fix the bug. For example, the most suitable developer might not have fixed the bug because they were unavailable to do so when the bug was discovered. As another example, the most suitable developer might not have fixed the bug because a novice developer was given the bug-fixing task for learning purposes.

6.2 Future Work

As future work, *DR_BEV* should be tested on other software projects. It should also be tested on a dataset of security vulnerabilities, where the buggy code coverage was produced

with a vulnerability detection technique. Also, other Information Retrieval techniques, such as topic modeling, should be experimented with. Lastly, in order to evaluate the usage of *DR_BEV* by developers, a user application should be developed to easily deploy *DR_BEV* on a new software project.

Bibliography

[1] Tricentis, "Software Fail Watch - 5th Edition," pp. 1–37, 2018.

[2] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?," *Acta Informatica Medica*, vol. 21, no. 1, pp. 63–66, 2013.

[3] K. lab, "Damage Control: the Cost of Security Breaches It Security Risks Special Report Series," 2015.

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," *Proceedings - International Conference on Software Engineering*, vol. 2006, pp. 361–370, 2006.

[5] H. Wu, H. Liu, and Y. Ma, "Empirical study on developer factors affecting tossing path length of bug reports," *IET Software*, vol. 12, no. 3, pp. 258–270, 2018.

[6] G. Jeong, S. Kim, and T. Zimmermann, "Img-929155833-0001.Pdf," pp. 111–120, 2009.

[7] A. Lamkanfi, J. Pérez, and S. Demeyer, "The eclipse and mozilla defect tracking dataset: A genuine dataset for mining bug information," *IEEE International Working Conference on Mining Software Repositories*, pp. 203–206, 2013.

[8] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," *Proceedings - 2013 International Conference on Availability, Reliability and Security, ARES 2013*, pp. 446–454, 2013.

[9] P. Liu, J. Su, and X. Yang, "Research on software security vulnerability detection technology," *Proceedings of 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011*, vol. 3, pp. 1873–1876, 2011.

[10] MeiJunjin, "An approach for SQL injection vulnerability detection," *ITNG 2009 - 6th International Conference on Information Technology: New Generations*, pp. 1411–1414, 2009.

[11] M. K. Gupta, M. C. Govil, and G. Singh, "An approach to minimize false positive in SQLI vulnerabilities detection techniques through data mining," *2014 International Conference on Signal Propagation and Computer Technology, ICSPCT 2014*, pp. 407–410, 2014.

[12] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities.," *Ndss*, 2003.

[13] M. F. Ringenburg and D. Grossman, "Preventing format-string attacks via automatic and efficient dynamic checking," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 354–363, 2005.

[14] D. Kergl, "Enhancing Network Security by Software Vulnerability Detection Using Social Media Analysis Extended Abstract," *Proceedings - 15th IEEE International Conference on Data Mining Workshop, ICDMW 2015*, pp. 1532–1533, 2016.

[15] F. Servant and J. a. Jones, "W HOSE F AULT : Automatic Developer-to-Fault Assignment Through," *Proceeding ICSE '12 Proceedings of the 34th International Conference on Software Engineering*, no. 2, pp. 36–46, 2012.

[16] V. Akila, G. Zayaraz, and V. Govindasamy, "Bug triage in open source systems: a review," *International Journal of Collaborative Enterprise*, vol. 4, no. 4, p. 299, 2014.

[17] D. G. Lee and Y. S. Seo, "Systematic review of Bug report processing techniques to improve software management performance," *Journal of Information Processing Systems*, vol. 15, no. 4, pp. 967–985, 2019.

[18] T. Weissman, "Bugzilla," 1998.

[19] Atlassian, "JIRA," 2002.

[20] GitHub, "GitHub Issue Tracker," 2002.

[21] Pilato, C. Michael, Ben Collins-Sussman and B. W. Fitzpatrick, *Version control with subversion: next generation open source version control*. O'Reilly Media, Inc, 2008.

[22] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, "Version Control with Subversion (r4543)," vol. 7, 2011.

[23] The CVS Team, "CVS - Open Source Version Control," 1990.

[24] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang, "Detecting vulnerabilities in C programs using trace-based testing," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 241–250, 2010.

[25] W. G. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for NEutralizing SQL-injection attacks," *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, pp. 174–183, 2005.

[26] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pp. 467–478, 2016.

[27] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings - International Conference on Software Engineering*, pp. 467–477, 2002.

[28] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," *16th Int. Conference on Software Engineering and Knowledge Engineering*, pp. 92–97, 2004.

[29] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using Chinese bug data," *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pp. 451–455, 2009.

[30] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson, *Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts*, vol. 21. Empirical Software Engineering, 2016.

[31] S. R. Lee, M. J. Heo, C. G. Lee, M. Kim, and G. Jeong, "Applying deep learning based automatic bug triager to industrial projects," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. Part F1301, pp. 926–931, 2017.

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, pp. 1–12, 2013.

[33] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," *ACM International Conference Proceeding Series*, pp. 171–179, 2019.

[34] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (BTS) based on latent semantic indexing and support vector machine," *4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009: Simposio para Estudantes de Doutoramento em Engenharia de Software*, pp. 216–221, 2009.

[35] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage,"

Proceedings - International Computer Software and Applications Conference, pp. 576–581, 2011.

[36] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining," *Journal of Software*, vol. 8, no. 9, pp. 2185–2190, 2013.

[37] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," *IEEE International Conference on Software Maintenance, ICSM*, 2010.

[38] S. Wang, W. Zhang, Y. Yang, and Q. Wang, "DevNet: Exploring developer collaboration in heterogeneous networks of bug repositories," *International Symposium on Empirical Software Engineering and Measurement*, pp. 193–202, 2013.

[39] W. Zhang, S. Wang, and Q. Wang, "KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity," *Information and Software Technology*, vol. 70, pp. 68–84, 2016.

[40] S. Xi, Y. Yao, X. Xiao, F. Xu, and J. Lu, "An effective approach for routing the bug reports to the right fixers," *ACM International Conference Proceeding Series*, 2018.

[41] J. Huang and Y. Ma, "Predicting the Fixer of Software Bugs via a Collaborative Multiplex Network: Two Case Studies," pp. 469–488, 2019.

[42] S. Q. Xi, Y. Yao, X. S. Xiao, F. Xu, and J. Lv, "Bug Triaging Based on Tossing Sequence Modeling," *Journal of Computer Science and Technology*, vol. 34, no. 5, pp. 942–956, 2019.

[43] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "DREX: Developer recommendation with K-nearest-neighbor search and EXpertise ranking," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp. 389–396, 2011.

[44] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," *Proceedings - International Conference on Software Engineering*, pp. 25–35, 2012.

[45] J. W. Park, M. W. Lee, J. Kim, S. W. Hwang, and S. Kim, "CosTriage: A cost-aware triage algorithm for bug reporting systems," *Proceedings of the National Conference on Artificial Intelligence*, vol. 1, pp. 139–144, 2011.

[46] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "DRETOM: Developer recommendation based on topic models for bug resolution," *ACM International Conference Proceeding Series*, pp. 19–28, 2012.

[47] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," *IEEE International Working Conference on Mining Software Repositories*, pp. 22–30, 2013.

[48] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," *Proceedings - Working Conference on Reverse Engineering, WCRE*, pp. 72–81, 2013.

[49] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," *Proceedings - International Computer Software and Applications Conference*, pp. 97–106, 2014.

[50] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, "Improving Automated Bug Triaging with Specialized Topic Model," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 272–297, 2017.

[51] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," *Proceedings of the ACM Symposium on Applied Computing*, vol. 2, pp. 1767–1772, 2006.

[52] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009*, pp. 131–140, 2009.

[53] J. Helming, H. Arndt, Z. Hodaie, M. Koegel, and N. Narayan, "Semi-automatic assignment of work items," *ENASE 2010 - Proceedings of the 5th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 149–158, 2010.

[54] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging (NIER track)," *Proceedings - International Conference on Software Engineering*, pp. 884–887, 2011.

[55] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Assigning change requests to software developers," *Journal of software: Evolution and Process*, pp. 3–33, 2012.

[56] M. Linares-Vasquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?," *IEEE International Conference on Software Maintenance, ICSM*, pp. 451–460, 2012.

[57] N. K. Nagwani and S. Verma, "Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes," *International Conference on ICT and Knowledge Engineering*, pp. 113–117, 2011.

[58] R. Shokripour, Z. M. Kasirun, S. Zamani, and J. Anvik, "Automatic bug assignment using information extraction methods," *Proceedings - 2012 International Conference on Advanced Computer Science Applications and Technologies, ACSAT 2012*, pp. 144–149, 2012.

[59] M. K. Hossen, H. Kagdi, and D. Poshyvanyk, "Amalgamating source code authors,

maintainers, and change proneness to triage change requests," *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*, pp. 130–141, 2014.

[60] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 122–132, 2014.

[61] A. S. Badashian, A. Hindle, and E. Stroulia, "Crowdsourced bug triaging," *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*, pp. 506–510, 2015.

[62] M. B. Zanjani, H. Kagdi, and C. Bird, "Using developer-interaction trails to triage change requests," *IEEE International Working Conference on Mining Software Repositories*, vol. 2015-Augus, pp. 88–98, 2015.

[63] H. Yang, X. Sun, B. Li, and Y. Duan, "DR-PSF: Enhancing Developer Recommendation by Leveraging Personalized Source-Code Files," *Proceedings - International Computer Software and Applications Conference*, vol. 1, pp. 239–244, 2016.

[64] X. Peng, P. Zhou, J. Liu, and X. Chen, "Improving bug triage with relevant search," *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, pp. 123–128, 2017.

[65] Travis, "https://docs.travis-ci.com/user/code-climate."

[66] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, "Multi-objective test report prioritization using image understanding," *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 202–213, 2016.

[67] G. Varoquaux, L. Buitinck, G. Louppe, O. Grisel, F. Pedregosa, and A. Mueller, "Scikit-

learn," *GetMobile: Mobile Computing and Communications*, vol. 19, no. 1, pp. 29–33, 2015.

[68] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016.

[69] F. Chollet, "Keras," 2015.

[70] E. Loper and S. Bird, "NLTK: The Natural Language Toolkit," 2002.

[71] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, 2018.

[72] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pp. 437–440, 2014.

[73] V. Dallmeier, "iBUGS - Software Engineering Chair (Prof. Zeller) - Saarland University."

[74] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating & improving fault localization techniques," *Icse*, no. August 2016, 2017.

[75] K. Jarvelin and J. Kekalainen, "IR evaluation methods for retrieving highly relevant documents," *SIGIR Forum (ACM Special Interest Group on Information Retrieval)*, vol. 51, no. 2, pp. 41–48, 2000.

[76] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*, no. January, pp. 89–96, 2005.

[77] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.