

# Modeling Software Developer Expertise and Inexpertise to Handle Diverse Information Needs

Frank Lykes Claytor

Thesis submitted to the Faculty of the  
Virginia Polytechnic Institute and State University  
in partial fulfillment of the requirements for the degree of

Masters of Science  
in  
Computer Science and Applications

Francisco Servant, Chair  
Stephen H. Edwards  
Bert Huang

May 7, 2018  
Blacksburg, Virginia

Keywords: Expertise, Code Review, Bug Triage  
Copyright 2018, Frank Lykes Claytor

# Modeling Software Developer Expertise and Inexpertise to Handle Diverse Information Needs

Frank Lykes Claytor

(ABSTRACT)

Expert software developer recommendation is a mature research field with many different techniques being developed to help automate the search for experts to help with development tasks and questions. But all previous research on recommending expert developers has had two constant restrictions. First, all previous expert recommendation work assumed that developers only demonstrate positive expertise. But developers can also make mistakes and demonstrate negative expertise, referred to as inexpertise, and show which concepts they don't know as well. Previous research on developer expertise hasnt taken inexpertise into account. Another restriction is that all previous expert developer recommendation research has focused on recommending developers for a single development task or expertise need, such as fixing a bug report or helping with a change request. But not all expertise needs can be easily classified into one of these groups, and having different techniques for every possible task type would be difficult and confusing to maintain and use. We find that inexpertise exists, can be measured, and that it can be used to direct inspection effort to find potentially incorrect or buggy commits. Additionally we investigate how different expertise finding techniques perform on a diverse set of long and short expertise queries and develop new techniques that can get more consistent cross query performance.

# Modeling Software Developer Expertise and Inexpertise to Handle Diverse Information Needs

Frank Lykes Claytor

(GENERAL AUDIENCE ABSTRACT)

Expert software developers are a useful source of information. There have been many papers that research techniques for recommending expert developers for different tasks and questions. But all previous research on recommending expert developers has had two constant restrictions. First, all previous expert recommendation work assumed that developers only demonstrate positive expertise. But developers can also make mistakes and demonstrate negative expertise, referred to as inexpertise, and show which concepts they don't know as well. Another restriction is that all previous work on recommending expert developers has focused on recommending developers for a single development task or question. But not all expertise needs can be easily classified into one of these groups, and having different techniques for every possible task type would be difficult and confusing to maintain and use. In our first chapter we show that inexpertise exists, can be measured, and that it can be used to help identify potentially buggy or incorrect code. In the second chapter we investigate how different techniques for finding expert developers perform when evaluated on different kinds of expertise finding tasks to find which technique works well on multiples types of tasks.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Related Work</b>	<b>5</b>
2.0.1 The Expertise Needs of Developers . . . . .	5
2.0.2 Recommending Expert Developers . . . . .	6
2.0.3 Automated Bug Triaging . . . . .	6
2.0.4 Code Review . . . . .	8
<b>3 Can Developer Inexpertise be Modeled to Support Development Tasks?</b>	<b>9</b>
3.1 Motivations and Research Questions . . . . .	9
3.2 Motivating Example . . . . .	10
3.3 Study 1: Is there a set of concepts that reflect developer inexpertise?	12
3.3.1 Methodology . . . . .	12
3.3.2 Results . . . . .	17

3.4	Study 2: Is developer inexpertise more prevalent in contributions in which developers make mistakes? Is developer expertise more prevalent in contributions in which mistakes are not found? . . . . .	19
3.4.1	Methodology . . . . .	20
3.4.2	Results 2.1: Mistakes that cause code to be corrected in code review . . . . .	22
3.4.3	Results 2.2: Mistakes that introduce bugs . . . . .	24
3.4.4	Results 2.3: Mistakes that cause buggy code to be accepted in code review . . . . .	29
3.5	Study 3: Can we model and apply developer inexpertise to automate support for developer tasks? . . . . .	31
3.5.1	Methodology . . . . .	31
3.5.2	Automatic Prediction Techniques . . . . .	32
3.5.3	Assessing Technique Accuracy . . . . .	32
3.5.4	Results . . . . .	33
3.6	Threats to Validity . . . . .	36
<b>4</b>	<b>Can a Single Technique Address Diverse Expertise-finding Needs?</b>	<b>38</b>
4.1	Motivations and Research Questions . . . . .	38
4.2	Research Methods . . . . .	40
4.2.1	Research Subject . . . . .	40
4.2.2	Study Design . . . . .	40
4.2.3	Expertise Finding Techniques . . . . .	42
4.2.4	Information Sources to Learn Expertise . . . . .	47
4.2.5	Artifact-Specific Expertise Needs . . . . .	49
4.2.6	Artifact Agnostic Expertise Needs . . . . .	51
4.3	Results . . . . .	54

4.3.1	RQ1: Would existing expertise-finding techniques provide consistent accuracy for diverse software artifacts? . . . . .	54
4.3.2	RQ2: Could we adapt existing automated expertise-finding techniques to achieve more consistent and accurate recommendations for diverse software artifacts? . . . . .	57
4.3.3	RQ3: What accuracy would existing expertise-finding techniques or our adapted expertise-finding techniques provide for short free-form expertise queries? . . . . .	61
4.4	Discussion . . . . .	66
4.5	Threats to Validity . . . . .	67
4.5.1	Internal Validity . . . . .	67
4.5.2	External Validity . . . . .	68
<b>5</b>	<b>Conclusion</b>	<b>69</b>
	<b>Bibliography</b>	<b>71</b>
	<b>Appendix A: Recall@1-5 for Chapter 4 Evaluations</b>	<b>79</b>

# List of Figures

3.1	Difference between the submitted version final corrected version of a code change in Openstack. The developer showed inexpertise (made mistakes) using <i>Neutron</i> client . . . . .	11
3.2	Difference between the submitted version final corrected version of a later code change in Openstack. The developer again showed inexpertise (made mistakes) using the <i>Neutron</i> client . . . . .	11
3.3	Expertise and inexpertise vocabularies for the first example code change	15
3.4	Expertise and inexpertise vocabularies for the second example code change . . . . .	15
3.5	Developer expertise and inexpertise vocabularies for both example code changes . . . . .	16
3.6	We compare developer expertise and inexpertise as fuzzy set differences.	17
3.7	Shared and separate vocabulary between developer expertise and inexpertise . . . . .	18
3.8	Shared vocabulary between code changes accepted or corrected in code review and their developer's expertise or inexpertise . . . . .	23
3.9	Shared vocabulary between code changes with or without bugs found and their developer's expertise or inexpertise . . . . .	25
3.10	Shared vocabulary between code changes with or without bugs found and their developer's expertise or inexpertise — only for corrected changes in code review . . . . .	27

3.11	Shared vocabulary between code changes with or without bugs found and their developer’s expertise or inexpertise — only for accepted changes in code review . . . . .	28
3.12	Shared vocabulary between <i>accepted</i> code changes with or without bugs found and their <i>reviewer’s</i> expertise or inexpertise . . . . .	30
3.13	Results from using different sorting methods to direct code inspection effort during code review . . . . .	34
4.1	Results of RQ1 Evaluation . . . . .	55
4.2	Results of RQ2 Evaluation . . . . .	58
4.3	Results of RQ3 Evaluation . . . . .	62



# List of Tables

3.1	Recall and precision of direction inspection effort using different techniques to find changes that should be rejected by code review . . . . .	35
4.1	Characteristics of Expertise Query Evaluations . . . . .	54
4.2	MRR Results from evaluating expertise techniques using artifact specific expertise queries. . . . .	56
4.3	MRR Results from evaluating expertise techniques using new expertise sources and artifact specific expertise queries . . . . .	59
4.4	MRR Results from evaluating expertise techniques using short free-form expertise queries . . . . .	63
1	Recall@1-5 for Artifact Specific Q/A Site Queries . . . . .	80
2	Recall@1-5 for Artifact Specific Bug Report Queries . . . . .	81
3	Recall@1-5 for Artifact Specific Feature Request Queries . . . . .	82
4	Recall@1-5 for Artifact Specific Mailing List Queries . . . . .	83
5	Recall@1-5 for Artifact Specific Code Queries . . . . .	84
6	Recall@1-5 for Artifact Agnostic Q/A Site Queries . . . . .	85
7	Recall@1-5 for Artifact Agnostic Bug Report Queries . . . . .	86
8	Recall@1-5 for Artifact Agnostic Feature Request Queries . . . . .	87
9	Recall@1-5 for Artifact Agnostic Mailing List Queries . . . . .	88
10	Recall@1-5 for Artifact Agnostic Code Queries . . . . .	89



# Chapter 1

## Introduction

Expert developers are a useful source of information. Software developers have many different information needs and experts are the most commonly used source of information [30]. Additionally many information needs and tasks can't be fully automated, such as questions that can't be answered by looking at documentation or development artifacts or development tasks that require a human developer such as bug fixing and feature implementation.

Many researchers have developed techniques to recommend expert developers to help with development tasks. Based on the past actions of developers they can accurately model a developer's knowledge and recommend a developer with suitable expertise for a given task or information need. But past expert developer recommendation research has had two big restrictions: They only look at positive expertise, and they only create techniques that recommend experts for a single task.

Past expert finding techniques assumed that every action a developer makes demonstrates their expertise. Every change they make to a file or every bug report they close is used to model their expertise and learn what they know. But not every contribution a developer makes demonstrates positive expertise. Developers make mistakes. They introduce defects and write messy code. This demonstrates their inexpertise, the terms and concepts that they don't know as well.

The first chapter of this thesis investigates our first research question: Can developer inexpertise be modeled to support development tasks? Existing techniques assume that all developer contributions reflect expertise that developers gain from making such contributions. We refine this assumption by studying whether some developer

contributions reflect lack of expertise (inexpertise).

We investigate a large set of code changes that were made by a large number of developers from the OpenStack open source project. We use code review data to find where developers make mistakes and what terms they frequently use incorrectly.

We observed that the studied developers showed inexpertise for a subset of their vocabulary that was separate from the rest of their vocabulary. Then, we observed that this inexpertise vocabulary was more prevalent when developers made mistakes — for three different kinds of common coding mistakes. Conversely, developers’ expertise vocabulary was more prevalent in their code changes for which no bugs were found. All such differences were statistically significant. Finally, we observed that an automatic technique based on developer inexpertise provided support for a developer task (identifying code changes to correct during code review) that outperformed a random baseline technique. While our model of developer inexpertise was not found to significantly improve in the precision of code inspection, our results indicate that developer inexpertise can be useful and motivate future work to create more robust inexpertise models that can better predict future errors and support development tasks.

In the second chapter of this thesis we investigate our second research question: Can a single expert developer recommendation technique address diverse expertise-finding needs?

There has been a lot of effort put into researching expertise recommendation tools for software development. They tackle such diverse problems as ”Who should fix this bug?” [5] [40], ”Who should review this patch?” [60] [67], or ”Who should implement this change request?” [27]. Existing tools analyze a given bug report description or other development artifact to recommend a list of potential expert developers who could handle that task. Additionally most prior research focused on making specialized tools that recommend developers for a specific task.

It might not be practical for a software development team to have one tool for recommending experts for fixing bugs, and one tool for recommending developers for implementing features, etc. Furthermore not every expertise need of a developer can be labeled as some kind of development task. Past research has found that software engineers have a wide range of information needs [30] and having a tool for every single one is unfeasible. Having multiple recommendation tools to keep track of would increase maintenance effort. They require infrastructure to store expertise data and to obtain more training data. Having many techniques that all take different training data and use it in different ways would make upkeep difficult and confusing.

We first evaluate different combinations of expertise sources used in past research (code [40], bug reports [5], feature requests, Q/A site answers [48], and mailing list messages [12]) and expertise finding techniques used in past research (VSM[40], LSI [40], Time TFIDF[55], and a Naive Bayes Classifier[43]) on five different types of expertise queries (Q/A site questions, bug reports, feature requests, mailing list questions, commits). This evaluation shows us how already existing techniques would perform when used with a general purpose expert developer recommendation tool.

Techniques designed specifically for a single kind of expertise query may perform much worse for queries they were never meant to deal with. Thus in addition to testing techniques used in the past we also investigate the effectiveness of modeling developer expertise using commit messages and using all expertise sources at once. Both of these new expertise sources provide higher average accuracy across all expertise query types. While the techniques we create do not obtain the highest MRR for each individual query type, they do have more consistent performance than other techniques that perform well on one kind of expertise query and poorly on all others.

The expertise queries used in past research all dealt with specific development artifacts and required a large amount of textual information or metadata. But a general purpose expert recommendation tool shouldn't require development artifacts as input. A general purpose tool needs a more generalized input method. Additionally developers have many expertise needs and some can't be easily paired with development artifacts [30] [25]. Also having a large requirement for textual or other information could make using expert recommendation tools tedious or overly time consuming, especially for a novice developer who lacks the most expertise and thus needs these tools the most.

Thus we also evaluate different expert recommendation techniques on short artifact agnostic expertise queries. We take the expertise queries created using development artifacts, and extract the titles or most relevant words of the artifacts to create short search queries that approximate the kinds of queries developers would create when using a general purpose expert developer recommendation tool. By using short free-form queries we can determine how different techniques perform when given less textual information, which better approximates the queries developers would use with such a general purpose tool.

This thesis provides two main contributions:

- A novel study of how developer inexpertise can be modeled and leveraged to assist with development tasks

- An novel evaluation of different expertise sources and expertise finding techniques on a wide range of artifact-specific and artifact-agnostic expertise queries

These contributions build upon past expert developer recommendation research by investigating the limitations of prior work, the fact that developer inexpertise isn't modeled and that the current techniques are only intended for a single task, and showing how moving past them can lead to a more practical and useful methods of modeling developer expertise. This work can serve as motivation and a basis for future work to further develop new techniques that look at developer expertise in different ways to tackle software developers many expertise and information needs.

# Chapter 2

## Related Work

### 2.0.1 The Expertise Needs of Developers

Ko et al. [30] observed developers in a collocated workspace to see what information needs they had and how they met them. They found that coworkers are the most frequently sought after source of information. Coworkers were found to be an especially useful source of information that couldn't be found easily in documentation or software development artifacts, such as questions about design decisions or figuring out why a software bug occurred. Ko et al. also found that some information needs can't be automated because they deal with information can't be stored in documentation and thus require the help of an expert developer. They found that developers have a wide range of information needs, and that expert developers are an important source of information to meet these needs.

Johnson et al. [25] analyzed the comp.lang.tcl Usenet newsgroup to find what kinds of questions developers ask and what answers they get. They found that 67% of the questions asked were "Goal Oriented" and requested help with task specific goals. Additionally about half of the questions asked were unable to be answered by simply looking at documentation.

Sillito et al.[56] observed novice and industrial programmers to learn what questions they ask while performed change tasks. They identified 44 different questions and categorized them into 4 categories based on whether they were dealing with single points of interest or the connections or subgraphs between many different points. Sillito et al. also found that the current tools for assisting with change tasks weren't sufficient and that tool support for higher level questions was needed.

## 2.0.2 Recommending Expert Developers

There is a large body of research on using historical data on developers to model their expertise and assign them to the proper development task. One of the earliest work on this subject was Expertise Recommender, created by McDonald and Ackerman [41]. Expertise Recommender recommended developers for change requests by looking at the files that needed to be changed and recommending the developer who last changed those files, and recommended developers for tech support calls based on the calls they worked with previously. Mockus et al. developed the Expertise Browser [42], which modeled developer expertise based on how many times they made changes to different files.

Schuler and Zimmerman later developed the idea of usage expertise, which modeled developer expertise based on the methods they used and the code they wrote instead of using the methods, classes, or files they edited[50].

Riahi et al. used StackOverflow answers to model the expertise and users and recommend experts for new questions [48]. Campbell et al. used email messages to model expertise, though they didn't focus specifically on software developers [12].

Expert recommendation is also researched in regards to code review. Balachandran [7] developed ReviewBot, a reviewer recommending tool based on recent edits to the lines affected by the code change and nearby lines. Thongtanuam *et al.* [60] recommend reviewers that changed similar file paths to the reviewed change.

## 2.0.3 Automated Bug Triaging

Automated Bug Triaging is an area of software engineering expert recommendation research that focuses on finding experts for the development task of fixing bugs. Many researchers have developed a diverse array of techniques for finding the correct expert developers to fix a bug (*e.g.*, [3, 4, 9, 11, 13, 14, 16, 18, 21, 23, 26, 28, 35, 39, 46, 54, 59, 61, 62, 69]). As the size of projects increases, and as open bug repositories get more and more popular, the number of bugs reported for a project can grow dramatically. This makes bug triaging even more difficult, necessitating the development of an automated technique.

The techniques used by researchers to automatically assign bug reports to expert developers are useful for our research on a general purpose expert recommendation tool. There are many automatic bug triaging techniques that rely only on the textual



information contained in the bug reports and use it to identify relevant developers.

Murphy et al. [43] found favorable results by training a Naive Bayes classifier with the textual data of the bug reports that developer had previously fixed. Anvik et al. [5] researched a similar method using Support Vector Machines.

Jeong et al. [24] leverage "bug tossing" data, information on how bugs assigned to one developer get reassigned to another, to identify bug fixing and social patterns. They used this data to supplement to recommendations given by machine learning approaches and obtained better bug triaging accuracy. Xuan et al. [64] also sought to improve upon the machine learning techniques by developing a semi supervised bug triaging approach that labeled unassigned bug reports to increase the size of the training set. Multiple papers looked into improving the machine learning bug triaging approach by modeling latent topics with Latent Dirichlet Allocation or Latent Semantic Indexing [44] [58] [63] [2].

Matter et al. [40] modeled developer expertise using a different source. Instead of looking at the bug reports developers previously fixed they based their expertise on the code they wrote. The cosine similarity between developer's code vocabulary and the bug report was used to sort the developers based on their expertise related to fixing a bug.

Shokripour et al. [55] also used a developer's past commits as an expertise source for a technique called Time TFIDF. Instead of using cosine similarity to compare a developer's code vocabulary to the vocabulary of the bug report, they added up the number of times that each developer used each word of the bug report in their previously written code. Though instead of relying only on word counts, Shokripour et al. weighted the word counts using both TF-IDF and a weighting scheme based on how recently the developer had used each term.

All of these techniques assign developers to different development tasks, but they use entire artifacts, such as a bug report, as input. In this paper we evaluate how different expert recommendation techniques handle short artifact agnostic queries that only contain a few words, as opposed to using the full text of a bug report. An artifact agnostic expert recommendation tool would allow us to recommend developers for a diverse range of information needs while not putting a large burden on the users of the tool.

### 2.0.4 Code Review

The code review dataset used in Chapter 3 was created by Hamaski *et al.* [20] to analyze the code review roles of different open source projects. They used the dataset to perform a social network analysis of the Android Open Source Project [66] and using profiling techniques and code review data to identify hidden experts in the Android Open Source Project [33]. While extensive work has been done with this dataset, no research has yet been done on analyzing developer inexpertise in the changes in the dataset.

Beller *et al.* [10] studied what kind of changes occur between the submitted and corrected revision of a code change under review. They focused on the ConQAT and GROMACS open source projects and examined a small subset of reviewed changes from each. They looked at what changed between each revision and manually classified these changes using the classification scheme specified by Mäntylä and Lassenius [38]. They found that about 75% of changes done during the code review process were related to maintainability, while only 25% dealt with fixing functionality, such as adding features or fixing defects. We model developer inexpertise by using the changes that are made to commits during code review to find where developers make mistakes. Since code review is able to fix both software defects and non-functional errors this means using code review changes gives us a diverse number of developer errors to build inexpertise models from.

# Chapter 3

## Can Developer Inexpertise be Modeled to Support Development Tasks?

### 3.1 Motivations and Research Questions

Our goal in this chapter is to study whether developer inexpertise can be modeled from developer contributions, and whether we can use it to support developer tasks. We perform our study by dividing it to address four different research questions.

**RQ1:** Is there a set of concepts that reflect developer inexpertise?

We begin our experiments by studying whether a set of concepts that reflect developer inexpertise can be identified at all. The concepts for which a developer shows inexpertise would have to: (1) include the concepts for which she makes mistakes, and (2) be separate from the concepts for which mistakes were not found.

Thus, we perform a first experiment in which we identify the concepts with which developers deal by collecting the words (vocabulary) that they use in their code contributions. Then, we study whether the vocabulary used by developers when they made mistakes (inexpertise vocabulary) is separate from the vocabulary they used when mistakes were not found in their code (expertise vocabulary). We also study whether that separation happens across all developers, and the extent to which it happens.

- RQ2:** Is developer inexpertise more prevalent in contributions in which developers make mistakes?
- RQ3:** Is developer expertise more prevalent in contributions in which mistakes are not found?

An additional expectation for the set of concepts that reflect developer inexpertise is that they would be more prevalent when developers make mistakes than when mistakes are not found in their contributions. Conversely, developer expertise concepts would be more prevalent when no mistakes were found in their contributions than when they made mistakes.

In our second experiment, we study the answer to RQ2 and RQ3 by comparing developer expertise or inexpertise vocabulary with the vocabulary that they used in their code contributions. We investigate RQ2 and RQ3 for three different kinds of common developer mistakes.

- RQ4:** Can we model and apply developer inexpertise to automate support for developer tasks?

Besides studying whether developer inexpertise can be identified and modeled, an additional goal of this paper is to study whether such inexpertise model can be applied to automate support for developer tasks. Therefore, we perform a third experiment to study the potential usage of developer inexpertise models to provide support for a particular developer task: identifying code changes that will have to be corrected in code review. For this purpose, we create three novel techniques that provide automatic predictions for this task and we compare their accuracy with a baseline technique.

## 3.2 Motivating Example

In this section, we describe a running example to motivate our study. Existing techniques that identify developer expertise assume that developers gain expertise with all their contributions. We refine this hypothesis by studying whether developers also demonstrate inexpertise in their contributions.

We illustrate this idea in Figure 3.1 with code change *I6ef6f9a9e257104f676dde7ec26be98417ec70e0* from the OpenStack project. Figure 3.1 shows the difference between the submitted

```

__init__.py 58,58c59,60
< raise exceptions.NeutronClientNoUniqueMatch(
---
>raise exceptions.NeutronClientNoUniqueMatch(resource=resource,
>Name=name)

test_name_or_id.py 112,112a112
---
>Except exceptions.NeutronClientNoUniqueMatch as ex:

```

Figure 3.1: Difference between the submitted version final corrected version of a code change in Openstack. The developer showed inexpertise (made mistakes) using *Neutron* client

```

/nova/network/neutronv2/api.py 724,724c726,726
< else:
< LOG.exception(_('Neutron error showing floatingip %s') % id)
< raise
---
> else:
>     raise

```

Figure 3.2: Difference between the submitted version final corrected version of a later code change in Openstack. The developer again showed inexpertise (made mistakes) using the *Neutron* client

code change and its corrected version. We took this information from Openstack’s Gerrit code-review system.

Since Gerrit allows us to observe the changes that the code underwent in code review, we can also observe what parts of the code were correct, and which ones were amended. In other words, we can observe for which parts the developer showed expertise and for which ones he showed inexpertise. In this case, the developer showed inexpertise for the *Neutron* client, since he missed two parameters in a method call and the definition of an exception.

If we look at that developer’s previous changes, they made mistakes every time that the word “neutron” was involved. If the developer had access to this information, they may have checked over their code more carefully, since they have made past mistakes when “neutron” was involved. Furthermore, this developer also made a later change ([I27e44ffd33a73c88745de8fda8fd7c374435808a](#)) in which they made a mistake again regarding the same *Neutron* client (see Figure 3.2). In particular, the developer made a mistake throwing an exception with Neutron.

In this paper, we study whether — like in this example — developers show their inexpertise in their contributions, and whether their developer inexpertise can be modeled and analyzed to support developer tasks.

### 3.3 Study 1: Is there a set of concepts that reflect developer inexpertise?

In the first part of our evaluation, we study *RQ1: Is there a set of concepts for which developers show inexpertise?* To answer this question, we study whether a set of concepts (words) exists in developer contributions that shows their inexpertise and that is different from the concepts (words) that show developer expertise.

We perform this study by gathering the vocabularies that developers used in their code changes, and separating them into words that showed their expertise and words that showed their inexpertise. Then, we compare the expertise and inexpertise vocabularies individually for each developer.

#### 3.3.1 Methodology

We obtained the set of concepts for which developers show expertise or inexpertise by analyzing a software project’s code review archive — *e.g.*, Gerrit [1]. In code-review systems, developers write code changes and submit them for review [49]. Then, the code changes are checked by other core developers, who recommend corrections and improvements before the changes can be merged into the source code repository. As such, code-review systems contain valuable information about code that developers wrote both showing expertise — the code was *accepted* after review — and showing inexpertise — the code had to be *corrected* before being accepted. Furthermore, code review systems may reveal the occurrence of more varied defects than those found in bug reports. Mantyla et al. [38] found that the defects found by code review are 25% functional defects, bugs, and 75% are defects related to the maintainability of the code and documentation.

As such, for each studied code change, we compared the differences between its version that was *submitted* for review and its accepted *merged* version. We depict the *submitted* and *merged* versions of an example code change in Figure 3.1. In its submitted version, the developer added code for a new kind of exception, but left out some key parts. In order to get the change approved by code reviewers the developer had to revise her code and fix her mistakes. Comparing these two versions of the code allows us to observe the mistakes that developers made.

In this study, we extract expertise (see Section 3.3.1) and inexpertise vocabulary (see Section 3.3.1) from code changes (see Section 3.3.1) from the code-review system of

a popular real-world software project (see Section 3.3.1). Then, we aggregate the extracted vocabularies individually for each developer to build developer expertise and inexpertise models (see Section 3.3.1). Finally, we compare the differences between the developer expertise and inexpertise vocabularies (see Section 3.3.1) to see if there is a set of concepts that clearly represent developer inexpertise.

### Modeling Expertise from Code Changes

For a given developer, we modeled her expertise vocabulary as the set of words in her code changes that she used correctly. We posit that if a developer included a word in her submitted version of a code change, and then that word was still in the final version of the change that was accepted and merged after code review, then the developer used that word correctly. Thus, we include in a developer’s expertise vocabulary for a code change all the words that she included in the submitted version of the code change and that still appeared in the merged version of the same code change.

Additionally, we assigned a score to these words representing the number of times that they were used correctly. As such, we assign the expertise score for a word  $w$  in a code change  $c$  as the minimum between the number of times that the word appeared in the submitted version of the code change and the number of times that the word appeared in the merged version of the code change (see Equation 3.1). Figures 3.3 and 3.4 illustrate the result of applying Equation 3.1 over our example code changes from Section 3.2.

$$\text{exp}_{c,w} = \min \{ \text{count}_{c_{\text{submitted}},w}, \text{count}_{c_{\text{merged}},w} \} \quad (3.1)$$

### Modeling Inexpertise from Code Changes

We modeled a developer’s inexpertise vocabulary as the set of words in her code changes that were used incorrectly. We posit that if a word was corrected in code review, the developer was showing inexpertise for it. Thus a developer’s inexpertise vocabulary for a code change contains all the words that were included a different number of times in its submitted version than in its merged version.

The reasoning behind this approach is that if a word occurs in the submitted version a certain amount and it occurs fewer times in the merged version, then that word was

incorrectly included in some cases. Likewise, if a word occurs more in the merged version than the submitted version, then the developer did not use that word enough times, indicating inexperience with that concept.

We assign a score to each word in the inexperience vocabulary representing to the number of times that it was used incorrectly. We assign the inexperience score for a word  $w$  in a code change  $c$  as the absolute difference between the number of times that the word appears in the submitted version of the change and the number of times that the word appears in the merged version of the change (see Equation 3.2). Figures 3.3 and 3.4 illustrate the result of applying Equation 3.2 over our example code changes from Section 3.2.

$$\text{inexp}_{c,w} = \left| \text{count}_{c_{\text{submitted}},w} - \text{count}_{c_{\text{merged}},w} \right| \quad (3.2)$$

## Data Collection

We studied developer expertise and inexperience by analyzing the OpenStack Gerrit code-review dataset created by Hamasaki et al.[20]. OpenStack is a “cloud based operating system” that provides many software tools to automate and optimize the use of computational, networking, and storage resources [45]. OpenStack requires that all code changes submitted be put through code review before being merged with the main branch of the project.

We pre-processed the OpenStack Gerrit dataset by focusing our analysis in developers for whom we could collect reasonably-sized expertise and inexperience vocabularies, and in code changes for which we could capture both expertise-showing and inexperience-showing words. Thus, we analyzed only *merged* changes, *i.e.*, those that were eventually accepted in the source-code repository. Merged changes are the only ones from which we could extract both code that showed expertise — was accepted and that showed inexperience — was corrected. We then filtered out the bot accounts and developers that never submitted a change that was merged and were left with a dataset of 2,996 developers and 84,745 merged changes. These developers have a mean of 28.29 code changes and median of 4 code changes per developer.

From these merged changes, 34,470 of them were accepted as submitted, and 50,275 were rejected and then corrected before they were allowed to be merged into the code repository.



```
Expertise vocabulary:
{{neutron, 1}, {client, 1}, {unique, 1}, {match, 1}, {exception, 1} {raise, 1}}
Inexpertise vocabulary:
{{neutron, 1}, {client, 1}, {exception, 1}, {except, 1}, {resource, 2}, {name, 2}}
```

Figure 3.3: Expertise and inexpertise vocabularies for the first example code change

```
Expertise vocabulary:
{{raise, 1}, {else, 1}}
Inexpertise vocabulary:
{{log, 1}, {exception, 1}, {neutron, 1}, {error, 1}, {showing, 1}, {floatingip, 1}}
```

Figure 3.4: Expertise and inexpertise vocabularies for the second example code change

### Extracting Vocabulary from Code Changes

We model expertise and inexpertise by using the bag-of-words approach, which is a simplified representation of vocabularies as sets of words, that is used in natural language processing and ignores word order [51] [52]. Thus, for each one of our studied code changes, we extract the words in their source-code lines by tokenizing them, ignoring all whitespace, numbers, and special characters. We split multiword identifiers, *e.g.*, camel case, into their individual words. We also use Porter2 stemming to cut word suffixes and thus avoid different tenses or uses of the same word being treated as separate entities. Lastly, we ignore stop words, such as "if, is, because" to avoid overly common words from entering the expertise and inexpertise sets. All such practices are common in natural language processing [40][29].

We model the expertise and inexpertise vocabularies for a change as *fuzzy word sets*, where word scores represent the *membership value* for that word in the set [68]. We calculate word scores as described in Section 3.3.1 and Section 3.3.1. The reason for this is that some words appear in both the expertise and inexpertise vocabularies, but with different frequencies. With this fuzzy set model, we capture such different frequencies. Figures 3.3 and 3.4 show the expertise and inexpertise vocabularies for our example in Section 3.2.

### Modeling Developer Expertise and Inexpertise

We model the concepts for which developers showed expertise (or inexpertise) by aggregating the expertise (or inexpertise) vocabularies that we extracted from all the code changes that developers produced.

We use fuzzy vocabulary sets to model developer expertise and inexpertise. In these

```

Expertise vocabulary:
{{neutron, 0.95}, {except, 0.83}, {error: 0.875}, {raise, 0.8}, {port, 0.72}}
Inexpertise vocabulary:
{{neutron, 0.05}, {except, 0.17}, {error: 0.125}, {raise, 0.2}, {port, 0.28}}

```

Figure 3.5: Developer expertise and inexpertise vocabularies for both example code changes

fuzzy sets, each word that a developer used in her changes obtains a membership value that represents the ratio with which the word showed expertise (or inexpertise) over all the developer’s changes.

We define the expertise (and inexpertise) vocabulary  $\text{Exp}_d$  (and  $\text{Inexp}_d$ ) as all the words  $w$  in the developer’s vocabulary  $W_d$ , where each word obtains a membership value  $m(w)$  equal to its aggregated expertise (or inexpertise) score  $\text{Exp}_d$  (or  $\text{Inexp}_d$ ) for all the changes  $c_d$  that the developer performed  $C_d$  (as in Equations 3.3 and 3.4). Figure 3.5 illustrates the developer expertise and inexpertise resulting from running Equations 3.3 and 3.4 for both our example code changes from Section 3.2.

$$\begin{aligned}
 \text{Exp}_d &= \{w \in W_d \mid m(w) = \text{exp}_{d,w}\} \\
 \text{exp}_{d,w} &= \frac{\sum_{c_d \in C_d} \text{exp}_{c_d,w}}{\sum_{c_d \in C_d} (\text{exp}_{c_d,w} + \text{inexp}_{c_d,w})} \cdot 100
 \end{aligned} \tag{3.3}$$

$$\begin{aligned}
 \text{Inexp}_d &= \{w \in W_d \mid m(w) = \text{inexp}_{d,w}\} \\
 \text{inexp}_{d,w} &= \frac{\sum_{c_d \in C_d} \text{inexp}_{c_d,w}}{\sum_{c_d \in C_d} (\text{exp}_{c_d,w} + \text{inexp}_{c_d,w})} \cdot 100
 \end{aligned} \tag{3.4}$$

### Comparing Developer Expertise and Inexpertise

In order to observe whether separate inexpertise concepts exist for developers, we compare our modeled expertise and inexpertise vocabularies for developers. If most of a developers vocabulary is shared between her expertise and inexpertise vocabularies, that would indicate that we cannot clearly identify the concepts that she has inexpertise in. In the opposite case, a strong vocabulary separation would indicate the existence of distinct concepts for which the developer showed inexpertise. We conceptually depict our comparison in Figure 3.6.

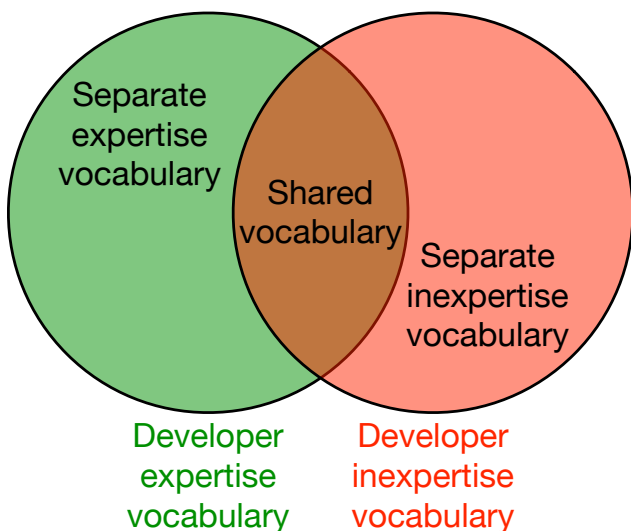


Figure 3.6: We compare developer expertise and inexpertise as fuzzy set differences.

For each developer  $d$ , we compute her developer expertise  $\text{Exp}_d$  and inexpertise  $\text{Inexp}_d$  and we compare them by using Equations 3.5, 3.6, and 3.7. Note that, since we modeled developer expertise and inexpertise as fuzzy sets, we use fuzzy set operations [68].

$$\text{Separate\_inexpertise\_vocabulary}_d = \frac{|\text{Inexp}_d - \text{Exp}_d|}{|\text{Exp}_d \cup \text{Inexp}_d|} \cdot 100 \quad (3.5)$$

$$\text{Separate\_expertise\_vocabulary}_d = \frac{|\text{Exp}_d - \text{Inexp}_d|}{|\text{Exp}_d \cup \text{Inexp}_d|} \cdot 100 \quad (3.6)$$

$$\text{Shared\_expertise\_vocabulary}_d = \frac{|\text{Exp}_d \cap \text{Inexp}_d|}{|\text{Exp}_d \cup \text{Inexp}_d|} \cdot 100 \quad (3.7)$$

### 3.3.2 Results

We present the results of this study in Figure 3.7. For each studied developer, we display a bar that represents her vocabulary percentages that are: separate to the inexpertise vocabulary (at the bottom), separate to the expertise vocabulary (in the middle), and shared between both vocabularies (at the top).

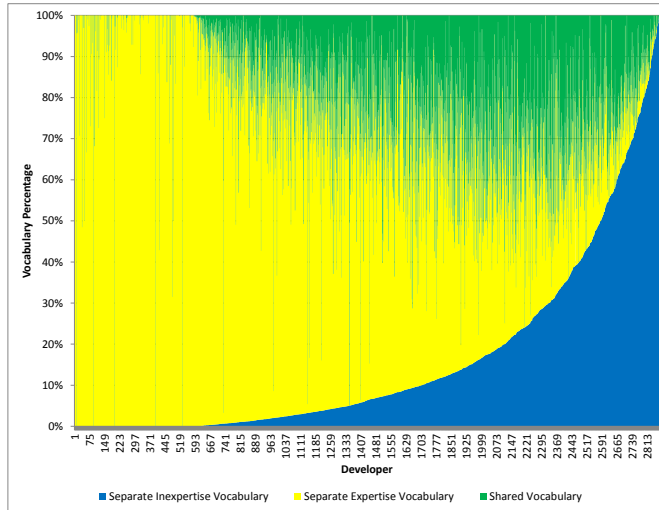


Figure 3.7: Shared and separate vocabulary between developer expertise and inexpertise

This figure shows a clear separation between expertise and inexpertise concepts — a median of 82.6% of developer vocabulary is separate to expertise or inexpertise vocabulary.

In particular for the inexpertise vocabulary, it represented a median 11.6% of developer vocabulary. We expected to observe a relatively low percentage for separate inexpertise vocabulary, since we also expected developers to write considerably more code that is correct than incorrect. Moreover, a median 11.6% size shows promise for the inexpertise vocabulary to be of enough size to allow us to make automatic inferences about developer inexpertise to support developer tasks. We study such promise in Sections 3.4 and 3.5.

### **3.4 Study 2: Is developer inexpertise more prevalent in contributions in which developers make mistakes? Is developer expertise more prevalent in contributions in which mistakes are not found?**

After we observed evidence for the existence of concepts for which developers showed inexpertise in Section 3.3, we now study these two research questions: *RQ2: Is developer inexpertise more prevalent in contributions in which developers make mistakes?* and *RQ3: Is developer expertise more prevalent in contributions in which mistakes are not found?* If the answer to these questions was positive, such evidence would show promise for building automatic predictors to support developers in detecting coding mistakes early.

We study three different kinds of coding mistakes. First, we study mistakes that would not pass the code-review filter, *i.e.*, that would cause code to have to be corrected in the code-review phase. Automatic predictors for this kind of mistakes could expedite the code-review phase by helping developers to avoid such mistakes.

Second, we study mistakes that introduced bugs in the code. Mistakes in the code review phase are mistakes that one would want to avoid, but they are mistakes that may or may not lead to software bugs. In fact, many of the problems and mistakes identified and fixed by code review are based on maintainability or documentation [10]. Therefore, we also separately study mistakes that caused bugs.

Finally, while developer inexpertise may cause developers to make mistakes when writing code, it may also cause them to make mistakes when reading and evaluating code. If a developer has inexpertise for the concepts involved in the code change, then she may miss a bug and allow the buggy code to be merged in the project. Thus, we also study mistakes that cause developers to mistakenly evaluate code changes in code review — in particular, by accepting code changes that contain bugs. These kind of mistakes are particularly dangerous, since they cause bugs to go unnoticed for a longer time, therefore increasing the cost of fixing them. Finding promising evidence for enabling automatic predictors for this kind of mistakes would be potentially highly impactful.

By validating our inexpertise model on three different types of developer errors we can more definitively show if there is a relationship between developer mistakes and

developer inexpertise.

### 3.4.1 Methodology

To carry out our experiment, we collect code changes with either of the three mistakes that we want to study, as well as code changes in which mistakes were not found. Then, we compare the vocabulary in these code changes with the involved developer's expertise and inexpertise vocabularies to study the prevalence of such vocabularies in code with or without found mistakes. Finally, we report the results of these comparisons, separately for each kind of mistake.

#### Data Collection

We performed the data collection process that we described in Section 3.3.1. For all collected changes, we analyze in this study their version that was submitted for code review (not the corrected one). We labeled the collected code changes into three different categories. Some code changes could be in multiple categories, *e.g.*, buggy code change corrected in code review.

**Code changes with no mistakes found.** Those that were accepted as submitted (24,508 code changes).

**Code changes with mistakes that caused them to be corrected in code review.** Those that were corrected before they were merged into the code repository (30,223 code changes).

**Code changes with mistakes that introduced a bug.** Those that introduced code that was later modified in a bug fix (see Section 3.4.1) (6,109 code changes).

#### Identifying Code Changes that Introduced a Bug

We identify code changes that introduce bugs by first identifying code changes that fix bugs, and then tracing the history of the fixed lines of code back to their origin.

We first identify all the bug-fixing changes in our studied dataset as all those that mention a bug ID in their commit message. Our subject software OpenStack has commit message guidelines that require bug-fixing commits to specify the bug-repository ID number of the bug being addressed.

Then, we apply the SZZ algorithm [57] (running GIT BLAME) to identify the originating code change for the lines of code that were removed or changed in the bug fix. We marked as bug-introducing any code change that originated any of such lines. Note that each line may have originated in a different code change. We identified a total of 15,053 bug-introducing code changes, out of which 11,837 were corrected by code reviewers, and 3,216 had their first submitted version accepted.

### Comparing Code Change Vocabulary with Developer Expertise and Inexpertness Vocabulary

In order to understand the relationship between the concepts involved in code changes and those that developers have expertise or inexpertise about, we compare code-change vocabularies to expertise and inexpertise vocabularies.

We model the expertise and inexpertise vocabularies individually for each developer, as we described in Section 3.3.1. For this experiment, we model developer expertise and inexpertise from the code changes that a developer made in the 90 days previous to the code change with which they are going to be compared. This ensures that the shared vocabulary percentage between a code change and the fuzzy vocabulary sets doesn't use any expertise or inexpertise data that was created after the code change was made. Thus, the developer expertise and inexpertise vocabularies are slightly different for each code-change comparison. We do this to faithfully represent developer expertise and inexpertise at the moment when the compared code change happened — without considering expertise and inexpertise observed either after the compared code change or too long before it.

We model the code change vocabulary as a bag of words that contains all the words used in its *submitted* version. We process the words in a code change as we described in Section 3.3.1.

We observe the prevalence of the developer expertise or inexpertise vocabulary in a code change vocabulary by measuring their shared vocabulary. In particular, we calculate the percentage of the considered code-change vocabulary that is shared with the expertise or inexpertise vocabulary. We apply Equation 3.8, where  $V$  is the other vocabulary that we compare with the code-change vocabulary. Since we modeled developer expertise and inexpertise as fuzzy sets — *i.e.*, we use fuzzy set operations [68], we consider the code change vocabulary also as a fuzzy set where all elements have a membership value of 1. when we compare the vocabulary of a code change to the two fuzzy vocabulary sets conceptually we are measuring how much

of the changes vocabulary belongs to either set. If a code change has a high shared vocabulary percentage with a given fuzzy vocabulary set then it contains many terms that have high membership values with that fuzzy set.

$$\text{Shared\_code\_vocabulary}(V) = \frac{|\text{Code\_vocabulary} \cap V|}{|\text{Code\_vocabulary}|} \cdot 100 \quad (3.8)$$

### 3.4.2 Results 2.1: Mistakes that cause code to be corrected in code review

In this first part of our second study we focus on mistakes that cause code to be corrected in code review. We study whether a developer’s inexpertise vocabulary is more common when this kind of mistakes are made than when they are not. We also study whether a developer’s expertise vocabulary is more common when this kind of mistakes are not made than when they are. If developers use more of their inexpertise vocabulary when they make mistakes that will be corrected in code review, this information may help us to automatically predict and warn them about such mistakes.

For this goal, we compute the code-change vocabulary for the submitted version of all code changes that were corrected in code review, and also of all those that were accepted. We also compute the expertise and inexpertise vocabulary for the developer that made the change, individually for each code change. In total, we compute and compare a total of 164,193 vocabularies (54,731 code-change vocabularies, plus their corresponding expertise and inexpertise vocabularies).

We present the results of these comparisons in Figure 3.8. This figure contains four violin plots, showing the distribution of shared vocabulary between (accepted and corrected) code changes and (expertise and inexpertise) developer vocabulary. We also include in this figure the p-values obtained by applying the two sided Mann Whitney U test — since the population of results was not normally distributed. We checked for statistical significance in the difference of shared vocabulary percentages that we observed between accepted and corrected code changes.

In these results, we observe that developers’ inexpertise vocabulary had a higher shared vocabulary with code changes that were corrected in code review (median 16%) than with code changes that were accepted (median 10%). We also observe that developers’ expertise vocabulary had a higher shared vocabulary with code changes



3.4. STUDY 2: IS DEVELOPER INEXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH DEVELOPERS MAKE MISTAKES? IS DEVELOPER EXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH MISTAKES ARE NOT FOUND? 23

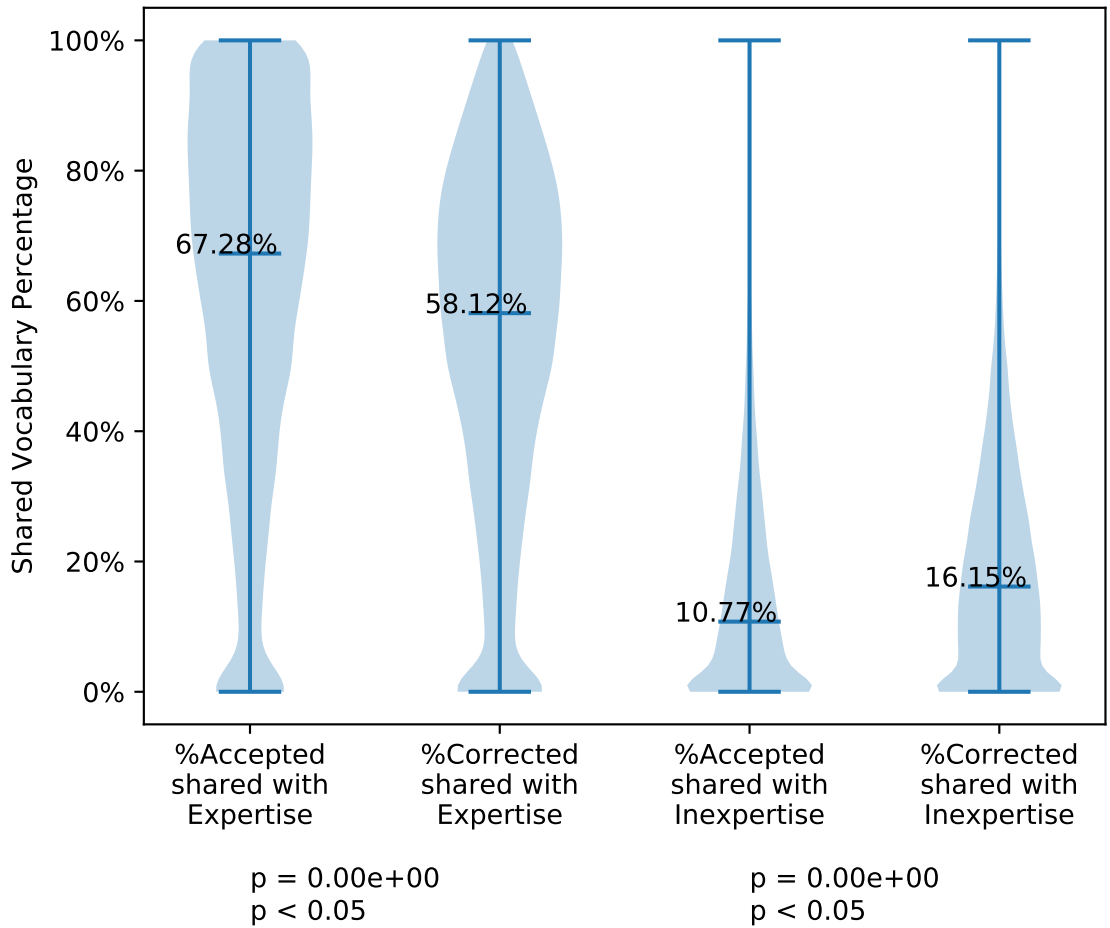


Figure 3.8: Shared vocabulary between code changes accepted or corrected in code review and their developer's expertise or inexpertise

that were accepted in code review (median 67%) than with code changes that were corrected (median 58%). Both of these differences were statistically significant ( $p < 0.05$ ).

Summing up, words that developers often used correctly were more common in accepted code changes than in corrected ones, and words that developers often used incorrectly were more common in corrected code changes than in accepted ones.

### 3.4.3 Results 2.2: Mistakes that introduce bugs

The second kind of mistake that we study is the introduction of bugs in code. We study whether inexpertise vocabulary is more common for code changes that contain a bug, and whether expertise vocabulary is more common for changes for which no bug was found. If developers use more of their inexpertise vocabulary when they make bug-introducing mistakes, this information may help us to automatically predict such bug-introducing changes.

For this study, we identify code changes that introduced bugs as described in Section 3.4.1. Then, we compare developer expertise or inexpertise vocabulary with the vocabulary in code changes with or without bugs found, as described in Section 3.4.1.

We display the results of this study in Figure 3.9. Similarly to Figure 3.8, we display four violin plots representing the shared vocabulary between code with or without bugs found and expertise or inexpertise vocabulary, as well as the p-values resulting from Mann Whitney U tests for statistical significance of such differences.

This figure shows that developers used more of their inexpertise vocabulary when they introduced bugs in their code (median 17%) than when they did not (median 13%). They also used more of their expertise vocabulary (median 61%) when bugs were not found in their code (median 59%). All these differences were statistically significant ( $p < 0.05$ ).

These results show the same trends that we observed for mistakes in code review. In other words, developer inexpertise vocabulary was more common both when developers got their code corrected in code review and when it contained bugs.

However, these results may have been caused by the fact that a larger number of code changes that contained bugs had also been corrected in code review. To study the effect of the code review outcome in these results, we also display our results from Figure 3.9 separately for corrected and for accepted code changes in Figures 3.10 and

3.4. STUDY 2: IS DEVELOPER INEXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH DEVELOPERS MAKE MISTAKES? IS DEVELOPER EXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH MISTAKES ARE NOT FOUND? 25

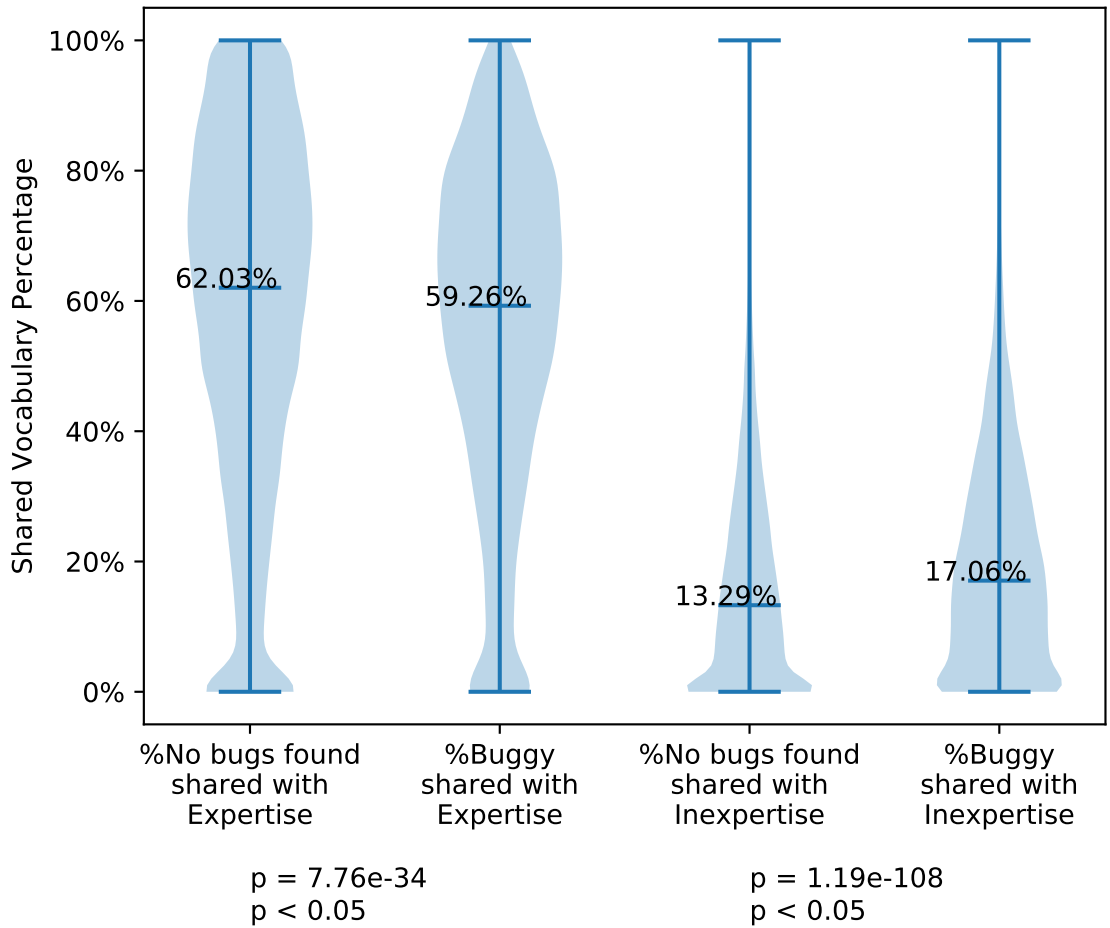


Figure 3.9: Shared vocabulary between code changes with or without bugs found and their developer's expertise or inexpertise

3.11, respectively.

When we divide the code changes that contained bugs into accepted and corrected changes, we still observe the inexpertise vocabulary being more common when bugs were introduced and expertise being more common when bugs were not found — in all cases and with statistical significance ( $p < 0.05$ ).

If we focus on the shared vocabulary between code changes and developer inexpertise, we observe in Figure 3.10 a lower difference between its median for code changes with no found bugs (16%) and code changes with bugs (18%) than what we observed in Figure 3.9 (13% to 17%). This result reinforces our observation that developer inexpertise was more common when mistakes were made. Since corrected code changes still contain mistakes (even if they are not bugs) — *e.g.*, maintainability or documentation mistakes [10], the developer inexpertise shared vocabulary would not have decreased much for code changes that had no bugs found but that were corrected in code review.

For accepted code changes (Figure 3.11), we observed a gap between the median shared inexpertise overlap for code changes with no found bugs (11%) and those with bugs (13%) that was also smaller than what we observed for all changes (13% to 17%). This effect reveals that the higher prevalence of developer inexpertise vocabulary for code changes with bugs that we observed in Figure 3.9 was in fact more strongly caused by a higher prevalence of inexpertise vocabulary in accepted code changes with bugs when compared to accepted code changes with no bugs found — instead of being mainly caused by the larger number of code changes with bugs within corrected code changes.

Summing up, we observed that the shared vocabulary between a code change and her developer’s inexpertise vocabulary increased for changes in the following order: accepted code changes with no bugs found, accepted code changes with bugs found, corrected code changes with no bugs found, and corrected code changes with bugs found. We also observed that the shared vocabulary between a code change and her developer’s expertise vocabulary increased in the following order: corrected code changes with bugs found, corrected code changes with no bugs found, accepted code changes with bugs found, and accepted code changes with no bugs found.

Overall, all our observations for Figures 3.10 and 3.11 reinforce our earlier observations for Figures 3.8 and 3.9 that inexpertise vocabulary was more prevalent for both mistakes that caused code to be corrected and for mistakes that caused bugs. Furthermore, they also reinforce our earlier observations that expertise vocabulary was more prevalent when no mistakes were found (either causing corrected code or

3.4. STUDY 2: IS DEVELOPER INEXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH DEVELOPERS MAKE MISTAKES? IS DEVELOPER EXPERTISE MORE PREVALENT IN CONTRIBUTIONS IN WHICH MISTAKES ARE NOT FOUND? 27

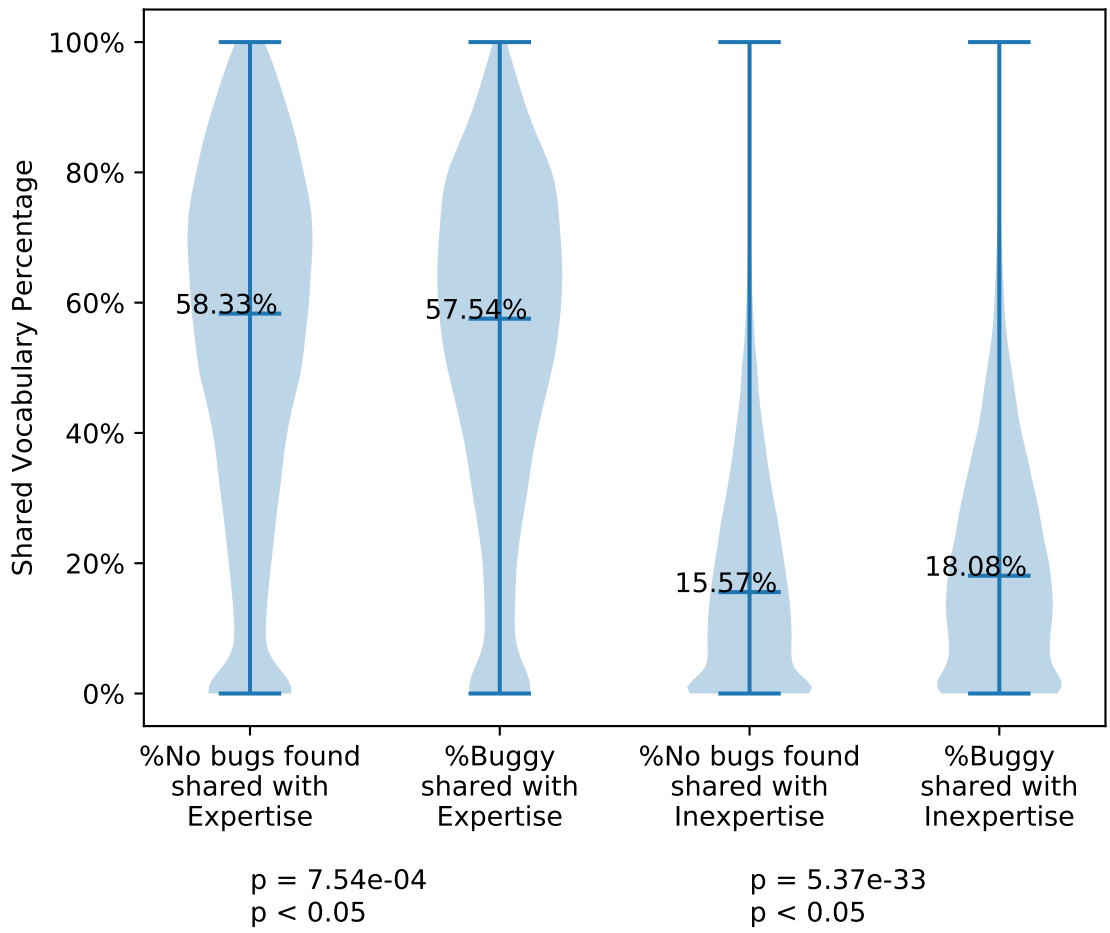


Figure 3.10: Shared vocabulary between code changes with or without bugs found and their developer's expertise or inexpertise — only for corrected changes in code review

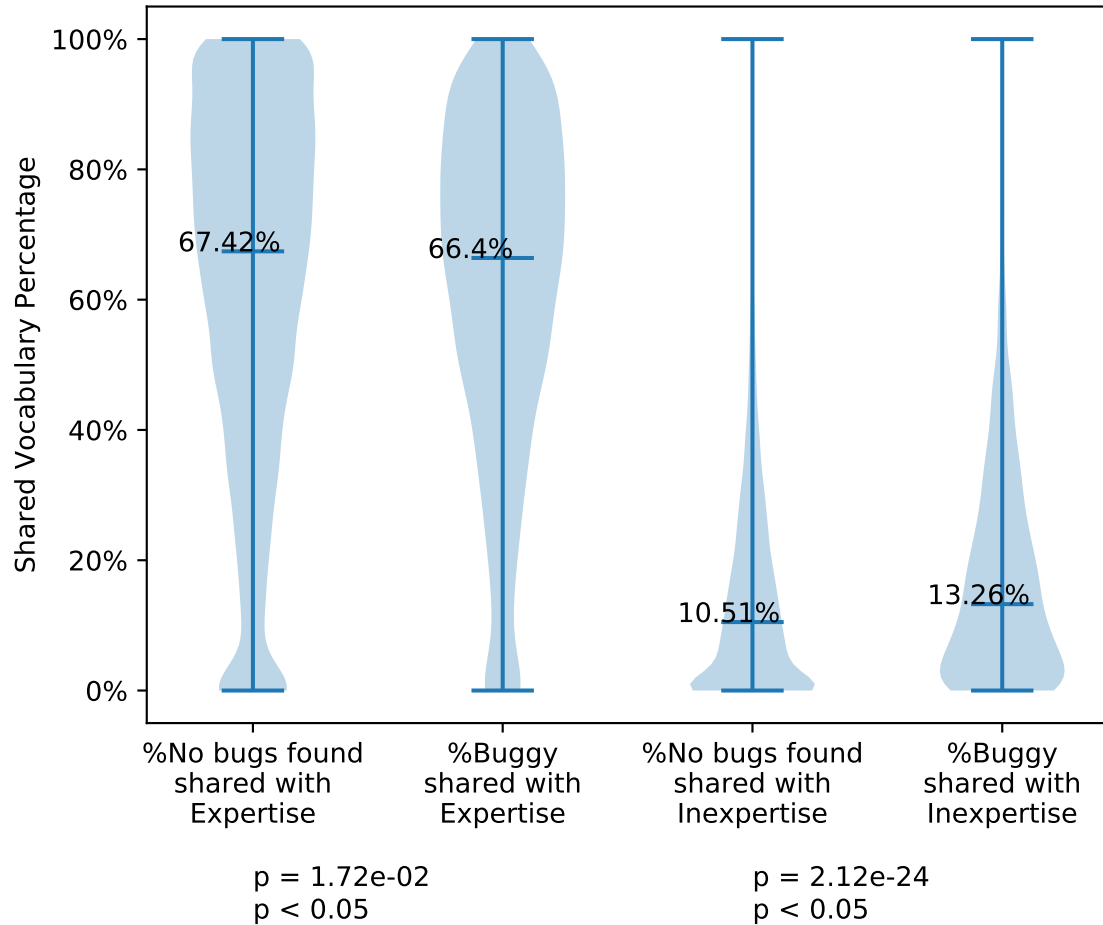


Figure 3.11: Shared vocabulary between code changes with or without bugs found and their developer's expertise or inexpertise — only for accepted changes in code review

bugs).

### 3.4.4 Results 2.3: Mistakes that cause buggy code to be accepted in code review

We study an additional kind of mistake: those that developers make when evaluating code written by others. While our other kinds of studied mistakes happen when developers *write* code, these mistakes happen when developers *evaluate* code. If we observed developer inexpertise vocabulary being more prevalent when they make mistakes reviewing code, it would show promise for automatically detecting such mistakes.

For this part of our study, we compare the expertise or inexpertise vocabulary of a developer with the vocabulary of accepted code changes *that she reviewed* — since we now want to assess mistakes in evaluating code. We collect data and compare vocabularies as described in Sections 3.4.1 and 3.4.1, respectively.

Note that we only study accepted code changes, in order to study mistakes of accepting a code change that contained a bug. However, we do not study corrected code changes, since we cannot automatically assess whether correcting code changes with no bugs found was a mistake — there are other kinds of mistakes that could have been corrected, *e.g.*, maintainability mistakes.

We display the results of this study in Figure 3.12. This figure also includes the results of running the Mann Whitney U test, as we did in Sections 3.4.2 and Sections 3.4.3. We observe in this figure that the inexpertise vocabulary for reviewers had a higher overlap with the code changes accepted by them when the code changes had a bug (10.96%) than when no bugs were found in them (8.32%). When we observe the shared vocabulary with a developer’s expertise vocabulary we find no significant difference between buggy and nonbuggy code changes. The difference between the shared vocabulary percentage with the developer’s inexpertise vocabulary was statistically significant ( $p \leq 0.05$ ).

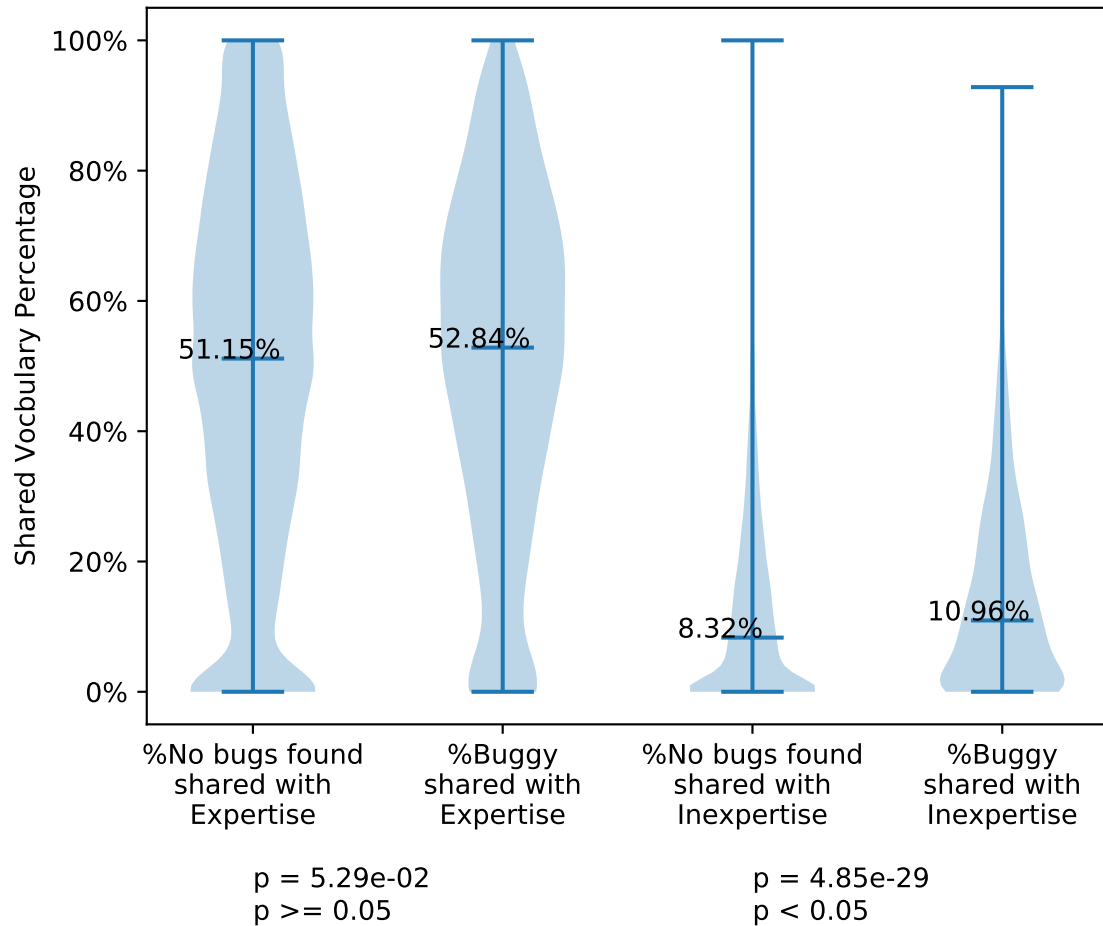


Figure 3.12: Shared vocabulary between *accepted* code changes with or without bugs found and their *reviewer's* expertise or inexpertise



## 3.5 Study 3: Can we model and apply developer inexpertise to automate support for developer tasks?

After having observed that developers have a separate vocabulary that shows developer inexpertise (see Section 3.3) and that developer inexpertise is more prevalent when developers make mistakes (see Section 3.4), we now study whether the developer inexpertise model can be applied to automate support for developer tasks.

We study the particular developer task of deciding which code changes will be corrected in code review. Unfortunately, code review is a time-consuming process, particularly when code changes are large or when the reviewer does not have enough expertise for them[6, 8]. Furthermore, the practice of code review is becoming more and more prevalent in software development, with many software projects now requiring that all their changes undergo code review. In such a scenario, developers would want to take an efficient approach to code review, prioritizing the review of code changes that will have to be corrected over those that will be directly accepted. The reason for this is that corrected code changes actually get sent back to the original developers for them to perform the corrections. Addressing first the code changes that will have to be corrected allows a larger parallelization of the development work — since it allows code-correction and code-review activities to happen in parallel. As such, developers would highly benefit from an automatic technique that predicts whether a code change will be corrected.

For this goal, we create novel automated techniques to support developers predicting corrected code changes in code review. These techniques make automatic predictions about whether a change will have to be corrected, based on our observations for RQ1, RQ2, and RQ3. We study the extent to which such techniques can provide accurate predictions.

We also perform a similar evaluation, but instead of identifying changes that will be rejected during code review we identify changes that will introduce bugs.

### 3.5.1 Methodology

We collect changes that were accepted or corrected in code review, as described in Section 3.4.1. Then, we use four different prediction techniques to obtain an

automatic recommendation for each code change as to whether it will be corrected in code review or introduce a bug. Finally, we assess the accuracy of such predictions.

### 3.5.2 Automatic Prediction Techniques

For this study, we created three different prediction techniques to automatically recommend whether code changes will be corrected in code review, according to the insights that we observed Studies 1 and 2. To the extent of our knowledge, no automatic technique exists yet to automatically predict whether code changes will be corrected in code review.

**Inexpertise.** This technique predicts that code changes are more likely to be corrected as their shared vocabulary percentage with their developer’s inexpertise increases.

**Expertise.** This technique predicts that code changes are more likely to be corrected as their shared vocabulary percentage with their developer’s expertise increases.

**Reverse Expertise.** This technique predicts that code changes are more likely to be corrected as their shared vocabulary percentage with their developer’s expertise decreases. The changes are sorted in the opposite order used by the Expertise technique.

**Line Count.** This technique predicts that code changes are more likely to be corrected as their size increases.

**Inexp\*LC.** This technique predicts that code changes are more likely to be corrected as the product of their line count and their inexpertise shared vocabulary percentage increases.

### 3.5.3 Assessing Technique Accuracy

We assess the accuracy of our studied techniques in an effort-aware manner using the precision and recall metrics.

First, we run all four techniques for all changes, obtaining a prediction score for each one of them. Then, we obtain for each technique a ranked list of the code changes in terms of their prediction score. That ranked list, from top to bottom, represents the order in which the technique recommends inspecting the code changes to find first those that will have to be “corrected”.

For the *Inexpertise* technique, we obtain a ranking of code changes in terms of their

shared vocabulary with their developer’s inexpertise vocabulary, from top to bottom in descending order — highest shared vocabulary first. For the *Expertise* and *Reverse Expertise* techniques, we obtain a ranking of code changes in terms of their shared vocabulary with their developer’s expertise vocabulary, from top to bottom in descending order — highest shared vocabulary first. For the *Line Count* technique, we obtain a ranking of code changes in terms of the number of lines they added. For the *Inexp\*LC* technique, we obtain a ranking of code changes based on the product of their line count and inexpertise shared vocabulary percentage, from top to bottom in descending order — highest product first.

Finally, we measure the precision and recall that the technique would provide if we spent a given percentage of the total effort required to inspect all changes, *i.e.*, the precision and recall obtained when inspecting only a given percentage of the top of the ranked list. Recall measures, out of all the changes that were actually “corrected”, what percentage was found within the inspected portion of the list. Precision measures, out of all the changes in the inspected portion of the list, what percentage was actually “corrected”.

We additionally compute the F1 score and the Receiver Operating Characteristic (ROC) curve for each technique.

The F1 score is the harmonic mean of the recall and precision of the technique.

$$F1Score = \frac{2 * Recall * Precision}{Recall + Precision}$$

The ROC curve graphs the relationship between the true positive rate and the false positive rate of the different techniques.

### 3.5.4 Results

We report the results of our third study in Table 3.1 in terms of precision and recall for different percentages of effort — by inspecting only a given top percentage of the ranked code changes, as described in Section 3.5.3.

We also report results in Figure 3.13, showing graphs of the recall, precision, and F1 score obtained at different levels of inspection effort. We also show the ROC curve for each technique. Additionally in these plots we show results for a random sorting method that would find X% of incorrect changes with X% of inspection effort. This allows for an easy baseline for comparison.

Table 3.1 shows that when we try to identify changes that should be rejected by code

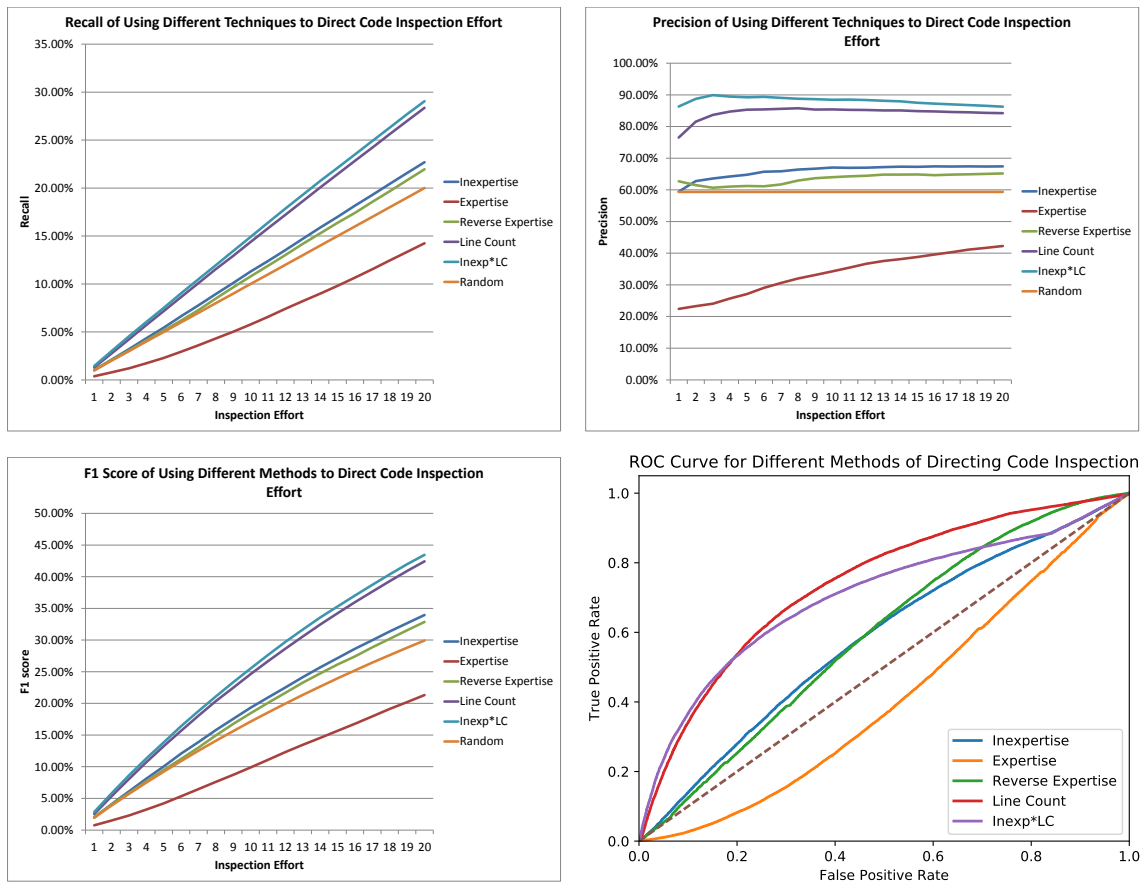


Figure 3.13: Results from using different sorting methods to direct code inspection effort during code review

Table 3.1: Recall and precision of direction inspection effort using different techniques to find changes that should be rejected by code review

Technique	Inexpertise		Expertise		Reverse Expertise		Line Count		Inexp*LC	
% Effort	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
1%	1.0%	59.43%	0.38%	22.41%	1.06%	62.74%	1.29%	76.53%	1.45%	86.32%
2%	2.11%	62.79%	0.78%	23.29%	2.07%	61.5%	2.75%	81.54%	2.99%	88.74%
3%	3.21%	63.58%	1.21%	24.05%	3.06%	60.67%	4.23%	83.69%	4.54%	89.9%
4%	4.33%	64.22%	1.73%	25.67%	4.11%	61.04%	5.71%	84.73%	6.03%	89.48%
5%	5.45%	64.78%	2.28%	27.11%	5.15%	61.2%	7.18%	85.31%	7.51%	89.23%
6%	6.64%	65.74%	2.94%	29.06%	6.18%	61.14%	8.63%	85.4%	9.03%	89.35%
7%	7.77%	65.87%	3.61%	30.59%	7.28%	61.74%	10.09%	85.57%	10.5%	89.06%
8%	8.95%	66.42%	4.31%	32.02%	8.48%	62.93%	11.55%	85.77%	11.96%	88.77%
9%	10.11%	66.67%	5.02%	33.11%	9.65%	63.67%	12.94%	85.36%	13.44%	88.65%
10%	11.29%	67.03%	5.77%	34.26%	10.78%	63.99%	14.38%	85.42%	14.9%	88.45%
11%	12.4%	66.96%	6.57%	35.44%	11.9%	64.26%	15.8%	85.29%	16.39%	88.49%
12%	13.54%	67.0%	7.41%	36.68%	13.03%	64.45%	17.22%	85.22%	17.86%	88.37%
13%	14.71%	67.18%	8.21%	37.52%	14.19%	64.81%	18.63%	85.1%	19.3%	88.15%
14%	15.88%	67.34%	8.99%	38.11%	15.28%	64.81%	20.06%	85.07%	20.74%	87.95%
15%	17.0%	67.29%	9.81%	38.82%	16.38%	64.85%	21.44%	84.88%	22.11%	87.52%
16%	18.17%	67.42%	10.66%	39.57%	17.41%	64.62%	22.84%	84.75%	23.51%	87.23%
17%	19.28%	67.35%	11.54%	40.3%	18.56%	64.84%	24.22%	84.58%	24.9%	86.98%
18%	20.44%	67.43%	12.46%	41.11%	19.67%	64.89%	25.62%	84.51%	26.31%	86.79%
19%	21.56%	67.37%	13.34%	41.68%	20.82%	65.05%	26.99%	84.33%	27.7%	86.56%
20%	22.7%	67.4%	14.25%	42.29%	21.96%	65.2%	28.36%	84.21%	29.05%	86.25%
100%	100.0%	59.37%	100.0%	59.37%	100.0%	59.37%	100.0%	59.37%	100.0%	59.37%

review, we get increased precision by sorting the changes by their inexpertise score. 59.37% of the code changes should be rejected by code review, thus a random sorting would obtain a precision of 59.37%. Sorting using Inexpertise can obtain a precision of over 67%. Conversely, sorting using expertise lowers precision significantly, which makes sense since the inexpertise and expertise metrics are negatively correlated. Sorting using Reverse Expertise leads to similar recall and precision as using Inexpertise, which makes sense because the weight of a word in the fuzzy expertise set is equal to 1 minus the weight of the word in the fuzzy inexpertise set. Thus a high inexpertise weight for a word will always indicate a low expertise weight. The only time this isn't the case is for new words that aren't in either set, in which case they have a membership value of 0 with both fuzzy sets.

While sorting using Inexpertise is useful, sorting by line count obtains even higher precision, with a peak precision above 85%. This is because the bigger a code change is the more things it changes at the more chance there is for a mistake to occur. Sorting the changes using Inexp\*LC obtains the best precision and recall, with peak precision above 89%, compared to the 59% precision obtain by a random sorting.

Additionally, the precision and recall for Line Count at 1% and 2% are much lower

than the precision of the other percentages. This means that the changes that are outliers with the very highest line count aren't necessarily incorrect. Inexp\*LC has 10% higher precision than Line Count when only looking at 1% of the total code changes.

Table 3.1 shows that modeling developer inexpertise can be useful for identifying code changes that contain mistakes that could cause a change to be rejected by code review.. While using inexpertise to direct inspection effort only does slightly better than random sorting, and using Inexp\*LC is only slightly better than using Line Count by itself, we have shown that considering inexpertise can lead to marginal improvements. This shows that modeling developer inexpertise can be useful, and future work can investigate creating more complex and useful models of inexpertise.

## 3.6 Threats to Validity

**Internal Validity:** We study developer mistakes as a uniform set, even though they could contain different mistakes, from security bugs to documentation problems. However, we classify developer mistakes into three different categories: in code review, introducing bugs, and evaluating code. We believe that having studied three diverse kinds of mistakes increases the trustworthiness of our results. Also, we used the SZZ algorithm to identify buggy code changes, which assumes that all lines changed by a bug fix change were buggy. While some imprecisions may have happened in this process, using the SZZ algorithm is a standard process for identifying bug-originating lines, *e.g.*, [29, 47, 53].

The results of RQ4 showed that large code changes are much more likely to require corrections during code review than smaller changes. This means that since we calculate developer inexpertise based on code review corrections then developers that make many large changes will have a lot more inexpertise data than those that do not. This could make very active experts seem like they have less expertise than less active developers if only the fuzzy vocabulary sets are considered, but this isn't an issue since we never compared the expertise or inexpertise vocabulary of two different developers. Additionally this shows that there could be other factors besides a developer's inexpertise that could lead to code corrections being made.

Our method of modeling developer inexpertise doesn't catch all possible errors or mistakes, only those that are detected during code review and only those that involve a word being added or removed from a code change. If words are simply rearranged

or lines are swapped around then no inexpertise will be detected.

External Validity: We performed our study over one open-source project. However, we performed our study over a very large number of code changes (84,745) and developers (2,996) in a largely popular open-source project (Openstack). These large numbers allow for a large diversity of observed code changes, which increases the likelihood of our results generalizing to other codebases.

# Chapter 4

## Can a Single Technique Address Diverse Expertise-finding Needs?

### 4.1 Motivations and Research Questions

In this chapter we investigate three main research questions relating to using expertise finding techniques to recommend developers for multiple types of tasks and information needs.

**RQ1: Would existing expertise-finding techniques provide consistent accuracy for diverse software artifacts?**

Past expert recommendation techniques were made with only a single task in mind. They work well for their given task, but they may be too specialized to work well with a general purpose technique. Typically recommendation techniques get their expertise queries and expertise sources from the same development artifacts. So they recommend experts for Q/A site questions by looking at their Q/A site answers. Or they recommend experts for bug reports by looking at past bug report activity, though automatic bug triage research has also looked into using code to model a developer bug fixing expertise. RQ1 takes techniques used in past research that have been tested with a single query type and sees how they perform on multiple kinds of queries.

**RQ2: Could we adapt existing automated expertise-finding techniques to achieve more consistent and accurate recommendations for diverse soft-**



**ware artifacts?**

We expand on the results of RQ1 by also evaluating using new techniques not used in past research. A general purpose expertise finding technique would need to work well for all query types. We investigate using new sources to model developer expertise, such as the commit messages they make or using all available sources at once. We evaluate all combinations of expertise sources and expertise finding techniques to see what works best for all queries. The answer to RQ2 will allow us to find an expert recommendation technique that does well for all query types and isn't overly specialized.

**RQ3: What accuracy would existing expertise-finding techniques or our adapted expertise-finding techniques provide for short free-form expertise queries?**

The answer to RQ2 shows that there is a technique that can recommend developers to work on multiple types of development artifacts. But the technique still requires the full text of software development artifacts as input. This large input size requires a lot of effort from the user, especially for expertise needs that don't correspond with an already existing artifact. We handle this by create artifact agnostic expertise queries that are generated from our expertise queries from RQ1 and RQ2. The new queries are short and not formatted like a specific development artifact so they better approximate the search queries users of a general purpose expert recommendation tool will write.

An artifact agnostic expert recommendation tool, one that doesn't require a full bug report, feature request, or Q/A site question before it can give results, would be very useful. It would allow developers to quickly obtain a list of relevant expert developers without the hassle of constructing large queries.

RQ3 takes the expertise finding techniques used for RQ1 and RQ2 and evaluates them on a collection of short queries generated from the large artifact specific queries used in the previous evaluations. Using short queries as input reduces the information the techniques have to work with. This could affect the different techniques in different ways. Thus the best technique found for RQ2 may not work as well when short queries are used.

## 4.2 Research Methods

To answer our research questions we need to take past expertise recommendation techniques and evaluate them on a diverse set of information needs. We create multiple sets of expertise queries from development artifacts and obtain the ground truth about which developer had expertise with the queries. We then evaluate many combinations of expertise sources and expertise finding techniques to find their accuracy across multiple sources. We then go further and evaluate recommendation techniques when they are only given short artifact agnostic queries that contain less textual information to use and observe how that changes their effectiveness. By exhaustively evaluating every possible combination of expertise source, expertise finding technique, and expertise query we can get a clear picture of how different techniques handle diverse information needs.

### 4.2.1 Research Subject

We selected OpenStack, a cloud computing open source project, as the subject system for our expertise research. OpenStack was selected because it has a large amount of data and different development artifacts that could be used to model developer expertise in different ways. Additionally focusing on making recommendations using a single project removes the biases that could result from using multiple projects. We create expertise queries from the contributions made to the OpenStack project from 7/31/2011 to 12/31/2014. This allows us to get data from a large range of time without causing evaluations to run too long. This also captures the time of peak activity from OpenStack's Q/A site, which was most active in 2013-2014 and then saw a heavy drop off in use.

### 4.2.2 Study Design

We evaluate each combination of expertise source type and expertise finding technique on the artifact specific and artifact agnostic versions of each of the five expertise query types. By testing all possible combinations we can determine how existing techniques perform on multiple query types, how they can be adapted to perform better, and how different expert recommendation techniques perform on short free form queries.

When evaluating a combination of expertise source type  $E$  and expertise finding technique  $T$  on a query  $Q$  we do the following steps for each expertise query.

1. Create candidate list of potential expert developers
2. Sort the list of developers based on their expertise with  $Q$  using technique  $T$  and expertise source  $E$
3. Record the best rank of a correct expert developer in the ranked list

The best ranks of correct developers for each query can then be combined to determine the accuracy of the different techniques.

### **Creating a Candidate List of Developers**

The dataset we use contains data on over 2000 OpenStack developers, but it wouldn't make sense to include all developers in every ranked list for each query. Some developers only start contributing to OpenStack partway through the dataset. And some developers never answer a query of a certain type, so they shouldn't be considered potential experts for those queries.

We use a simple filter to determine which developers are put into the initial candidate list. For each query evaluated, before any similarity scores are calculated a candidate list of potential expert developers is formed. This list contains all developers that have previously answered a expertise query of the same type in the past.

### **Ranking the List of Developers**

Once the candidate list is created it is sorted using the combination of an expertise source and an expertise finding technique that leverages the source type to estimate a developer's knowledge. The expertise finding techniques output a similarity score between a developer's expertise and the query, and this can be used to sort the developers in the candidate list.

### **Recording the Best Rank**

Once the list of potential expert developers is sorted we can obtain a score for the given query by finding the best rank obtained by a developer that the ground truth

has shown to be an actual expert on the query.

### Accounting for Time

We use historical software development data from the OpenStack project to build the developer expertise profiles and train the TF-IDF and LSI models. But while our dataset contains data from over 3 years of OpenStack development, it wouldn't make sense to rank developers for query made in 2012 using historical data from 2014.

Thus we take measures to make sure that no ranked lists of developers are created using data that wasn't available when the query was originally made. For each query we create each developer's expertise profiles using their contributions made within a year of the query.

Retraining the LSI, TF-IDF, and Naive Bayes classifiers for every query would take a prohibitively long time. We account for this by evaluating the queries in chronological order and retraining the LSI, TF-IDF, and Naive Bayes classifiers for each new month. The models are trained using a years worth of data and are never more than a month out of date.

### 4.2.3 Expertise Finding Techniques

Given a query and a set of software developers there are many different ways to rank the developers based on their estimated expertise with the query. We investigate 5 different methods to investigate how they handle the challenges of recommending developers for a diverse range of expertise needs. We evaluate 3 information retrieval techniques (VSM, LSI, VSM+LSI), 1 machine learning technique (Naive Bayes), and Time TFIDF, a bug triaging technique that leverages time metadata on a fine grained level to recommend developers [55].

Each expertise finding technique leverages a developer's expertise sources to output a similarity score between the expertise query and the developer's expertise. This score is used to rank the developers.

All of the expertise finding techniques we evaluate model developer expertise using the contributions they make to the OpenStack project, preprocess textual documents into word vectors, and reweight the word vectors using TF-IDF to make distinctive terms more important.

## Preprocessing Documents

Before a document, be it a commit, a bug report, or any other textual data, can be converted to a word vector it goes through several preprocessing steps. Words are extracted from the document using regular expressions, ignoring all numbers and special characters. Camelcase identifiers are split similar to how it was done in prior research [40] [16]. We also removed stop words, specifically the stop words specified in the NLTK Corpus. Lastly the terms are stemmed using the Porter2 stemmer.

Once all of these preprocessing steps are done the document has been converted to a list of terms. A word vector for the document can then be made based on how often each word appears in the list of terms.

## Building Developer Expertise Profiles

When using the expertise finding techniques we rank developers for an expertise query by comparing the vocabulary of their expertise profiles to the terms in the search query. The expertise profiles are built using the historical data of the developer's past contributions to the project. Each developer's expertise can be modeled using any of the expertise sources used in this paper.

The expertise profile for a developer is built using the contributions of the developer made within a year of when the query was made. The expertise sources made by the developer are preprocessed into word vectors. Then the vectors are added together to form the expertise profile for a developer, a word vector containing the count of how many times the developer used each term in the past year with a specific expertise source. We create expertise profiles based on a year of developer activity in order to get a large enough amount of data to be able to make informed decisions on a developer's expertise, but not such a large amount that irrelevant data from far in the past is used.

## TF-IDF

Term Frequency - Inverse Document Frequency (TF-IDF) is a way to reweight the word counts in word vectors to deemphasize words that appear in many documents, while emphasizing words that appear in only a few documents. This makes is so that the words that are the most distinct between documents have the most impact on the cosine similarity scores between documents. TF-IDF is frequently used for

automated bug triaging techniques that utilize textual information [55] [40] [5].

TF-IDF reweights the word counts of the developers' expertise profiles based on how many developers used the term previously in the code they wrote. If every developer used a certain word then it will have a very low weight. But if a word is distinctive to only a few developers then it is important and is given a higher weight.

### Naive Bayes

Machine learning is commonly used in automatic bug triaging techniques [5] [16]. Naive Bayes and Support Vector Machine classifiers are the most common machine learning techniques used. We decided to focus on evaluating Naive Bayes since it outperformed SVM in the evaluation done by Shokripour *et al.* when they tested Time TFIDF against different techniques [55].

Machine learning techniques used for bug triage handle the expertise sources differently than other expertise finding techniques. While all other expertise finding techniques we investigate combine expertise sources together to form large expertise profiles for developers, machine learning approaches instead leave expertise sources as separate word vectors and use them as training data. While past research on bug triage and machine learning only used bug reports as the training data, we investigate the applications of training a Naive Bayes classifier using other expertise sources as well.

Once the Naive Bayes classifier is trained on the expertise sources an expertise query can be converted to a word vector and the classifier can predict the probabilities that each developer has expertise with that query. The probabilities reported for each developer are used as the similarity score between their expertise and the expertise query.

### VSM

The Vector Space Model (VSM) expertise recommendation technique ranks developers on their expertise with a given query by sorting them by the cosine similarity between their expertise profiles and the expertise query. The developers' expertise profiles and the expertise query are preprocessed and reweighted using TF-IDF. After those steps the expertise profiles and expertise query are word vectors that can be easily compared using cosine similarity.

**Time Weighting** We also employ a temporal weighting technique to weight the cosine similarities given by the VSM technique. Previous work has also used temporal weighting with VSM to avoid recommending developers that have been inactive for a long period of time [40].

We weight the VSM similarity score between a developer and a query based on how many days it is between the developer’s last contribution using the current expertise source and the date the expertise query was made.

$$TimeWeight(D, Q, E) = \frac{(Q.date - latestDate(D, E)).days}{365}$$

Where  $latestDate(D, E)$  is the most recent date, at the time the query Q was made, that the developer had created an expertise source of type E.

## LSI

The Vector Space Model runs into issues when people use words with multiple meanings, or when they use different words that mean the same thing. When documents are represented only with word counts then comparing using cosine similarity would only find two documents similar to each other if they use the exact same words in the exact same way.

Latent Semantic Indexing (LSI) is meant to combat these problems by reducing the dimensions of the word vectors and finding the semantic relationships between words that occur together frequently [17]. LSI is could be useful for a general purpose software expert recommendation tool because the search engine is tested on short artifact agnostic expertise queries that don’t contain a lot of textual information. LSI has previously been used in bug triaging research [40] [2] [22].

An LSI model is created by taking a years worth of commits and converting them a document term matrix. Using Singular Value Decomposition the dimensions of the matrix can be reduced so instead of showing word counts it shows weights related to semantic topics. We reduce the dimensions of the corpus so it contains 300 LSI topics, which is a common number used in past research [32] [34]. Once the corpus is converted to semantic topics other word vectors can also be converted to the same topic space.

The LSI expertise finding technique adds an extra step to the VSM technique. Before the profiles and queries are compared using cosine similarity they are converted to the LSI topic space, reducing their dimensions. The resulting cosine similarity is

then temporally weighted similar to what we do with VSM.

### Time TFIDF

We also evaluate ranking algorithms based on a state of the art bug triaging technique, Time TFIDF, a technique developed by Shokripour *et al.* Time TFIDF gives developers a similarity score to a bug report based on how often they used terms found in the bug report in their previous commits, as well as using time metadata to account for how recently the developer used each of the terms in the bug report [55]. We take the Time TFIDF approach and apply it to multiple query types and expertise sources.

### VSM+LSI

The similarity score between a developer and a query outputted by VSM+LSI is simply the sum of the similarity score computed by VSM and the similarity score computed by LSI. When VSM+LSI is used as the expertise finding technique both the VSM and the LSI techniques are leveraged to obtain scores for each developer so they can be ranked based on their estimated expertise with the expertise query. VSM+LSI is meant to leverage the strengths of both VSM and LSI to get improved results.

### Evaluation Metric

We evaluate the different expert recommendation techniques by computing the Mean Reciprocal Rank of the rankings given to the correct developers. For each query the list of developers is ranked. The better the rank of the correct developers the better the technique is.

$$MRR = \frac{1}{Q} \sum_{i=1}^Q \frac{1}{Rank_i}$$

Where  $Q$  is the number of queries evaluated, and  $Rank_i$  is the rank of the top ground truth developer in the ranked list returned for the  $i$ th query.

Many expert recommendation papers report results as Recall@1-Recall@10. But since this paper deals with investigating 35 different expert recommendation techniques on 10 different sets of expertise queries reporting Recall@X results would



create too many data points. MRR captures the same trends but as a single data point and has been used in prior research [55].

In addition to using MRR we also use the Wilcoxon Ranked Sum Test to determine if there is a statistically significant difference between the rankings given by two techniques. Whenever we state that there is a statistically significant difference between the rankings of two techniques we mean that the Wilcoxon Ranked Sum Test had a p value of less than 0.05.

#### 4.2.4 Information Sources to Learn Expertise

Past techniques used different expertise sources to model developer expertise and to assign developers to different tasks. We want to develop an expertise finding technique that can recommend developers for many different expertise needs. Since prior research has been mostly focused on creating tools for recommending experts for only one specific type of task the expertise sources used by these tools might not be suitable for our more generalized expert recommendation tool. Thus we evaluate the expert recommending effectiveness of 6 different types of expertise sources: code, bug reports, commit messages, Q/A site answers, feature requests, and mailing list messages, as well as the effectiveness of using all six sources at once. We look at all examples of these expertise sources from the OpenStack project that were made within our studied time frame. The vocabulary and metadata of these different expertise sources are used to build the expertise profiles for the developers and model the terms that each developers has demonstrated expertise with.

##### Q/A Site

OpenStack uses Launchpad to host a Q/A site. We investigate using the vocabulary of the answers developers make on OpenStack's Q/A site to model their expertise. Riahi *et al.* evaluated using StackOverflow answers and different expertise finding techniques to recommend expert users for new questions [48].

##### Bug Reports

The vocabulary of the bug reports made on OpenStack's open bug repository on Launchpad are mined using the Launchpad API. The commit messages of OpenStack

bug fixing commits contain the bug id for the bug being fixed. These messages are used to link bug fixes and bugs together. Developer expertise is modeled using the title and description of the bug reports they fix. The bug reports fixed by developers are a commonly used source of developer expertise for bug triaging techniques [5] [16].

### Feature Requests

We investigate the use of modeling developer expertise using the title and description of the feature requests they implement. Developers put a feature request id in the commit message if they are implementing a feature and we use this to connect developers to the feature requests they implement.

### Mailing List Messages

Prior work has looked into using the contents and flow of email messages to model the expertise of users, though not specifically with software developers [12] [31]. The messages that developers sent on the OpenStack-dev mailing list were mined and used to model developer expertise. The vocabulary of the body of the message is used to model developer expertise and create an expertise profile.

### Code

We mine the commits made by the developers on the OpenStack GitHub repository to obtain the vocabulary of the code the developers write as one example of developer expertise. The code changes made by developers are a frequently used source of developer expertise in automated bug triaging research [40] [55] [50].

### Commit Messages

Commit messages are also mined from OpenStack's GitHub repository. No bug triaging technique has used commit messages as an expertise source, though Yang *et al.* [65] examined the characteristics of various GitHub projects and found that using commit messages as a source of expertise could be a useful direction for future research. Commit messages use natural language to describe the intent and effects

of a code change. Thus they may be more useful for modeling developer expertise in a way that can be easily compared to short natural language queries, as opposed to using code, which contains more words but also more noise.

### All Sources

We also evaluate leveraging every expertise source at the same time and treating each as the same type of source. Different expertise sources could be better at different query types so using all six at one time could produce better cross query results. By investigating the applications of using multiple sources at one time we can see what techniques work well when given multiple expertise sources with different semantics and vocabulary.

#### 4.2.5 Artifact-Specific Expertise Needs

A general purpose expert recommendation tool should work with a wide variety of expertise needs and query types.

From OpenStack we were able to obtain queries from 5 software development artifacts: Q/A Site Queries, Bug Report Queries, Feature Request Queries, Mailing List Queries, and Code Queries. These five query types cover a wide range of expertise needs, and by investigating which recommendation techniques work with which query type we can learn how best to construct an expertise search engine that can tackle many kinds of queries.

Each expertise query is a set of terms that express an information need that a developer has expressed expertise with. Expertise queries serve as the input for a general purpose software expertise recommendation tool that would then output a ranked list of potential expert developers. We also obtain the ground truth for each query, which is the names of the expert developers that answered or worked with the query.

**Q/A Site Queries** OpenStack utilizes the Answers feature of the Launchpad tool, which allows them to host a StackOverflow-like Q/A forum on their Launchpad page. From this forum users and developers can ask questions about OpenStack, and developers can answer them. The population of question askers is made up of mostly non-developer OpenStack users. This means that looking at Q/A site queries will

show how different techniques handle recommending developers for usage questions asked by people that aren't developers themselves.

A dataset of Q/A site queries is created by using OpenStack's Q/A site. We created a query for each answered question. Each question on the Q/A site has a short title and a longer body of text. The artifact-specific Q/A site query contains the terms in both the title and the body of the question.

The ground truth for a Q/A Site expertise query is the list of the developers that posted answers in response to the Q/A Site question.

Previous work has looked into recommending experts for new StackOverflow questions and other Q/A sites [48] [19] [36]. But these papers did not investigate using many different sources to recommend experts and also didn't use a Q/A site that was linked to a single open source project.

**Bug Report Queries** Bug triaging, the assignment of developers to open bug reports, is a very active research field and many researchers have proposed techniques to make finding experts for bug reports easier [15]. We create queries from the OpenStack bug reports by mining them from their Launchpad bug repository. The bug reports contain a short title and longer description, and the text from both are used to create the terms for the artifact specific bug report queries.

The ground truth for a bug report expertise query is the list of the developers that committed bug fixes that contained the bug report's bug id in the commit message.

**Feature Request Queries** Launchpad also supports a feature request repository, called blueprints. Queries are created from the feature requests the same way they are made for bug reports. The expertise queries contains the terms used in both the summary and description of the feature requests. Kagdi *et al.* [27] previously researched how to recommend developers for change requests.

The ground truth for a feature request expertise query is the list of the developers that made commits that contained the feature request's id in the commit message.

**Mailing List Queries** The openstack-dev mailing list was used to obtain a fourth type of query. Developers frequently use the mailing list to ask questions and get responses from expert developers, making it a useful source of potential search queries. But not all mailing list threads are asking for expertise.

A simple heuristic is used to identify the mailing list threads that are requesting the help of an expert developer. Any mailing list thread whose subject line contains any of the words "Who", "What", "When", "Where", "Why", "How", "Problem", "Bug", or the character "?", are assumed to be asking a query for an expert developer to answer.

From these querying mailing list threads we obtain expertise queries. The terms from the subject line and body of the mailing list messages are used to create the artifact specific mailing list expertise queries.

The ground truth for a mailing list expertise query is the list of the developers who responded to the querying mailing list thread.

Expertise queries similar to mailing list queries were previously researched by Krulwich *et al.* [31].

**Code Queries** We additionally generated search queries from the diffs of the commits made by the developers of the OpenStack project. Ma *et al.* [37] previously used commits as the testing set for evaluating the effectiveness of modeling implementation expertise vs usage expertise. The terms from the added lines in each commit's diff are used as the terms for the artifact specific code expertise queries.

The ground truth for a code expertise query is the developer that made the commit.

### 4.2.6 Artifact Agnostic Expertise Needs

An expert recommendation tool that works well simultaneously for multiple development artifacts would be useful, but it would still be dependent on the user generating the proper development artifact. If a user had an expertise need they would need to create a bug report, a q/a site question, or some other artifact to serve as input for the recommendation technique.

We create a dataset of artifact agnostic expertise queries that would treat all expertise queries the same and expect them all to be in the general format, a short natural language query. An expert recommendation tool that works with short artifact agnostic queries would require less time and effort to use and could be tested in a way that treats all expertise needs the same.

From our 5 sets of artifact specific queries we generate 5 set of artifact agnostic queries that are much shorter and approximate the types of queries a developer would use

with a general purpose expert recommendation tool. We generate short artifact agnostic expertise queries from development artifacts so that we can approximate the real queries developers would use with an expertise search engine (by using shortened versions of human written artifacts) and so that we can easily obtain the ground truth on who the expert developers for each artifact agnostic query are.

Since the short queries are generated from development artifacts, such as bug reports or Q/A site questions, they aren't truly artifact agnostic. They just approximate the types of short artifact agnostic search queries developers would write for a general purpose expert developer recommendation tool. Obtaining truly artifact agnostic queries would require a human study where developers in a software project would use a general purpose recommendation tool and input their own queries. But such a study would require an already working technique, it would take a long time, we would get less queries, and it would be difficult to find the correct ground truth developers. By automatically generating short queries from artifacts we can create a large number of queries that approximate the true artifact agnostic queries that developers would use, and we can use those queries to create a technique that works well with a general purpose expert recommendation tool that could be used in a future human study.

**Q/A Site Queries** Artifact agnostic Q/A site expertise queries are created using only the title of the Q/A site question.

- How to config multiple l3 agent for quantum?
- How to edit the SNAT rules of the L3 agent?
- How to create multiple table in tab in openstack horizon dashboard?

**Bug Report Queries** Artifact agnostic bug report expertise queries are created using only the title of the bug reports.

- Resource Group gets rebuilt when parameter is changed
- RandomString doesn't fail-fast when validating
- mongodb reconnect test is too slow
- Spark plugin doesn't work with Cinder volumes

**Feature Request Queries** Artifact agnostic feature request expertise queries are created using only the title of the feature requests.

Some examples of search queries created using OpenStack feature requests are:

- Merge OpenStack Puppet Modules
- Decouple nested stacks from their parent
- Shard management operations
- Add tests for keystoneclient python API interfaces

**Mailing List Queries** Artifact agnostic feature request expertise queries are created using only the subject line of the querying mailing list threads.

Some examples of search queries created using the OpenStack developer mailing list are:

- How to attach multiple NICs to an instance VM?
- Can gatekeeper middleware be removed from pipeline?
- Preserving ephemeral block device on rebuild?
- How does the libvirt domain XML get created?

**Code Queries** The artifact agnostic code expertise queries are generated by taking the vocabulary of the diff and shrinking it to only 5 terms.

To select the five terms we calculate how many times each term occurs in the diff of the commit. Then we weight the term frequency counts using TF-IDF and we select the terms with the top five TF-IDF weights.

The five selected words are combined to form an artifact agnostic code expertise query.

Some of examples of search queries created using the diffs of OpenStack commits are:

- guid opensus webhelp docbkx zypper

Table 4.1: Characteristics of Expertise Query Evaluations

Query Type	# of Queries	Median Candidate List Size	Median Answer Key Size	Median A-S Word Count	Median A-A Word Count
Q/A Site	1,243	169	1	63	5
Bug Reports	10,411	1,082	1	44	6
F. Requests	217	121	1	29	5
Mailing List	724	247	2	58	8
Code	81,470	1,650	1	25	5

- userinput prompt keystone chown var
- revdescript itemizedlist listitem orname remark

## 4.3 Results

### 4.3.1 RQ1: Would existing expertise-finding techniques provide consistent accuracy for diverse software artifacts?

We display in Table 4.2 the MRR results that we obtained for all our studied techniques, given all the combination of expertise sources and queries. In order to make it easier to observe the differences in accuracy provided by different techniques, sources and queries, we highlight the MRR results with different colors for different ranges: gray (0.00–0.09), red (0.01–0.19), orange (0.20–0.29), yellow (0.30–0.39), green (0.40–0.49), cyan (0.50–0.59), and teal (0.60–0.69). We did not observe MRR values higher than 0.69.

We also display the MRR results as bar graphs in Figure 4.1.

The best expert recommendation technique for each query type:

- Q/A Site: Q/A Site Answers and VSM+LSI
- Bug Reports: Bug Reports and VSM+LSI
- Feature Requests: Feature Requests and VSM+LSI
- Mailing List: Mailing List and VSM+LSI
- Code: Code and VSM



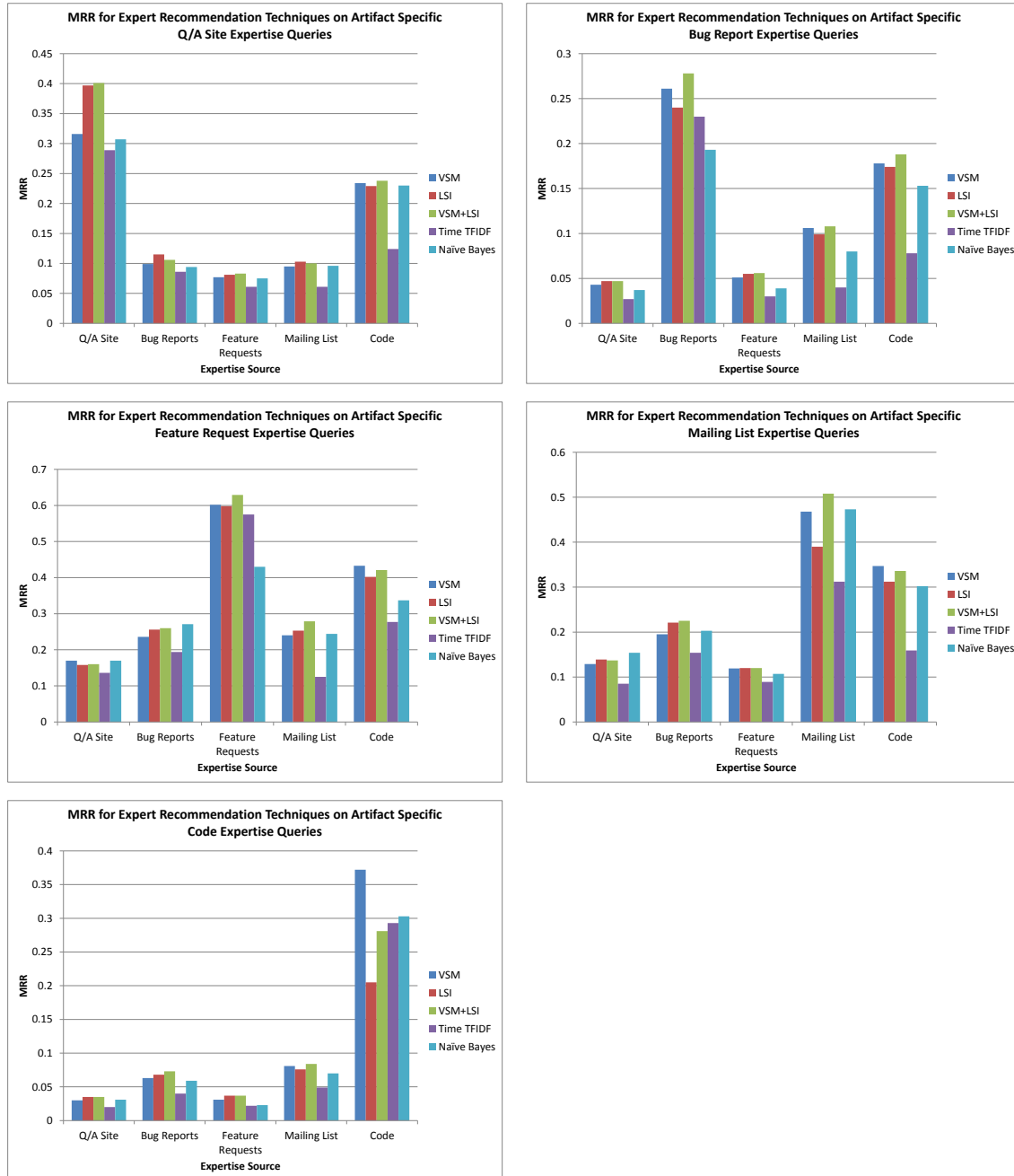


Figure 4.1: Results of RQ1 Evaluation

Table 4.2: MRR Results from evaluating expertise techniques using artifact specific expertise queries.

Expertise Source	Expertise Query	Naive Bayes	VSM	LSI	Time TFIDF	VSM+ LSI
Q/A Site	Q/A Site	0.307	0.316	0.397	0.289	0.401
Q/A Site	Bug Reports	0.037	0.043	0.047	0.027	0.047
Q/A Site	F. Requests	0.170	0.170	0.158	0.136	0.160
Q/A Site	Mailing List	0.154	0.129	0.139	0.085	0.137
Q/A Site	Code	0.031	0.030	0.035	0.020	0.035
Q/A Site	Average	0.140	0.138	0.155	0.111	0.156
Bug Reports	Q/A Site	0.094	0.099	0.115	0.086	0.106
Bug Reports	Bug Reports	0.193	0.261	0.240	0.230	0.278
Bug Reports	F. Requests	0.271	0.236	0.256	0.194	0.260
Bug Reports	Mailing List	0.203	0.195	0.221	0.154	0.225
Bug Reports	Code	0.059	0.063	0.068	0.040	0.073
Bug Reports	Average	0.164	0.171	0.180	0.141	0.188
F. Requests	Q/A Site	0.075	0.077	0.081	0.061	0.083
F. Requests	Bug Reports	0.039	0.051	0.055	0.030	0.056
F. Requests	F. Requests	0.430	0.602	0.598	0.575	0.629
F. Requests	Mailing List	0.107	0.119	0.120	0.089	0.120
F. Requests	Code	0.023	0.031	0.037	0.022	0.037
F. Requests	Average	0.135	0.176	0.178	0.155	0.185
Mailing List	Q/A Site	0.096	0.095	0.103	0.061	0.100
Mailing List	Bug Reports	0.080	0.106	0.099	0.040	0.108
Mailing List	F. Requests	0.244	0.240	0.253	0.125	0.279
Mailing List	Mailing List	0.473	0.468	0.390	0.312	0.508
Mailing List	Code	0.070	0.081	0.076	0.049	0.084
Mailing List	Average	0.193	0.198	0.184	0.117	0.216
Code	Q/A Site	0.230	0.234	0.229	0.124	0.238
Code	Bug Reports	0.153	0.178	0.174	0.078	0.188
Code	F. Requests	0.337	0.433	0.402	0.277	0.421
Code	Mailing List	0.302	0.347	0.312	0.159	0.336
Code	Code	0.303	0.372	0.205	0.293	0.281
Code	Average	0.265	0.313	0.264	0.186	0.293

One thing that is immediately apparent is that the optimal expertise source for a given query type is the one that comes from the same source as the query. Modeling developer expertise with bug reports works very well for assigning developers to bug reports. Modeling developer expertise using Q/A site answers is the best way to recommend developers for Q/A site questions. And so on. This makes sense, since in those situations the expertise sources and queries have a large amount of vocabulary overlap and are probably more likely to use terms in with the same semantics. Additionally the when the expertise source and query source match up then the time metadata may be more useful. Trying to use mailing list or bug report time metadata to help assign Q/A site questions may end up hurting results, since a developer's mailing list or bug report activity may not match up temporally with their Q/A site activity. Using an expertise source that isn't frequently used, such as feature requests, can also hinder accuracy because of the fact that if you only use one expertise source to model expertise then you will be ignoring all developers that don't work with that specific expertise source.

Additionally the best ranking technique for each query type is an IR technique. VSM+LSI is the best technique for four query types, while VSM is the best technique for Source Code queries. IR techniques seem to best at fully utilizing the data available in full textual descriptions and are useful for a wide variety of query types. While the optimal expertise source is different for all query types, VSM+LSI is consistently useful for the different queries.

But while using the same source for both developer expertise and queries leads to the best results, most expertise sources don't do well when used to recommend developer for queries of a different type. Feature requests aren't a useful expertise source for recommending developers for bug reports. And Q/A site answers aren't a good way to model developer expertise when making recommendations for mailing list queries.

### **4.3.2 RQ2: Could we adapt existing automated expertise-finding techniques to achieve more consistent and accurate recommendations for diverse software artifacts?**

Most expertise sources do well on one query type and poorly on all others. Thus we decided to evaluate using two new expertise sources: commit message and all sources at once. The MRR obtained from combining these sources with the different expertise finding techniques is found in Table 4.3.

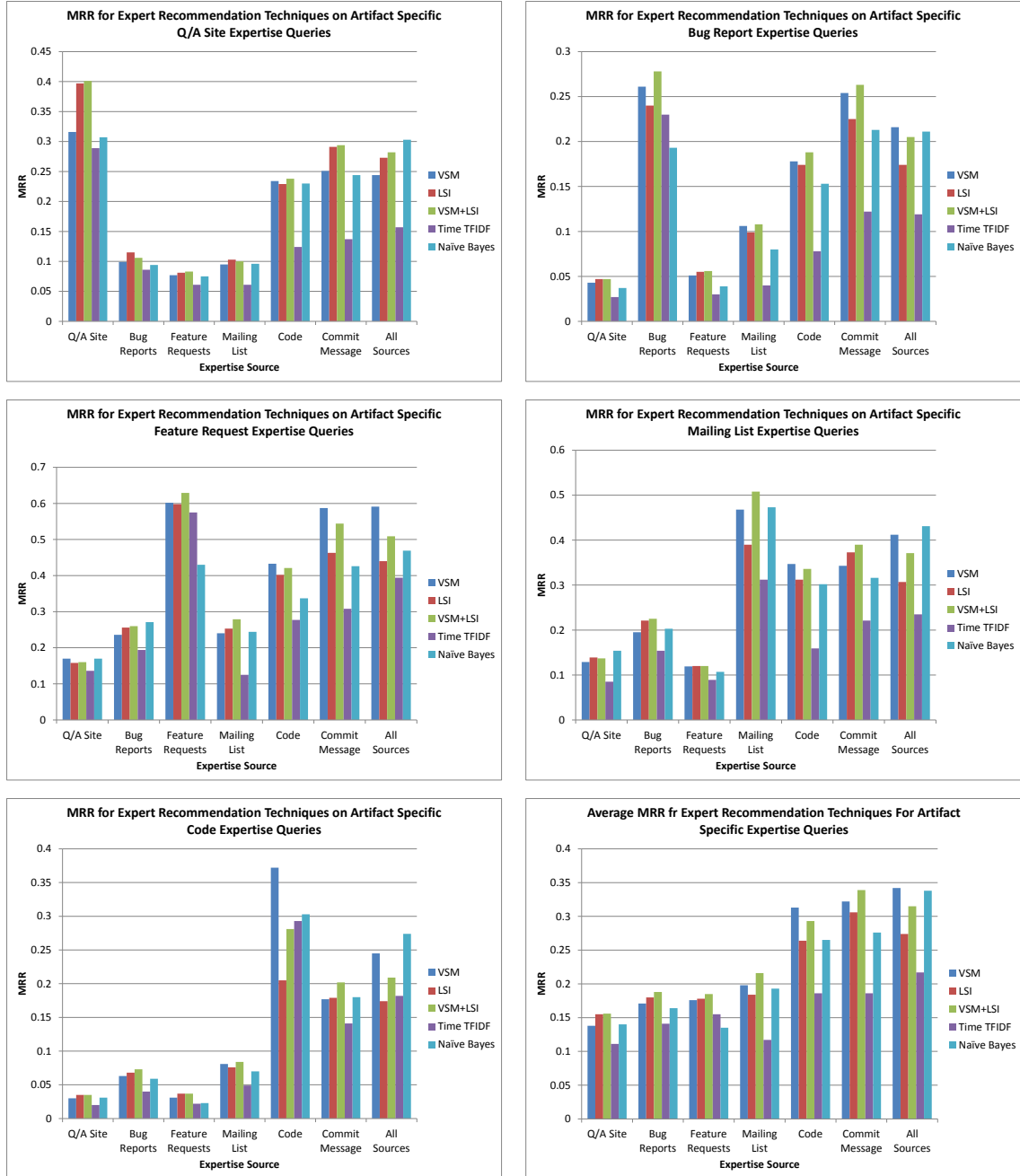


Figure 4.2: Results of RQ2 Evaluation

Table 4.3: MRR Results from evaluating expertise techniques using new expertise sources and artifact specific expertise queries

Expertise Source	Expertise Query	Naive Bayes	VSM	LSI	Time TFIDF	VSM+ LSI
Commit Message	Q/A Site	0.244	0.251	0.291	0.137	0.294
Commit Message	Bug Reports	0.213	0.254	0.225	0.122	0.263
Commit Message	F. Requests	0.426	0.587	0.463	0.308	0.544
Commit Message	Mailing List	0.316	0.343	0.373	0.221	0.390
Commit Message	Code	0.180	0.177	0.179	0.141	0.202
Commit Message	Average	0.276	0.322	0.306	0.186	0.339
All Sources	Q/A Site	0.303	0.244	0.273	0.157	0.282
All Sources	Bug Reports	0.211	0.216	0.174	0.119	0.205
All Sources	F. Requests	0.469	0.591	0.440	0.394	0.509
All Sources	Mailing List	0.431	0.412	0.307	0.235	0.371
All Sources	Code	0.274	0.245	0.174	0.182	0.209
All Sources	Average	0.338	0.342	0.274	0.217	0.315

We also display the MRR results as bar graphs in Figure 4.2. These graphs show the results of both the Rq1 and the RQ2 evaluations.

We observe that the optimal expert recommendation techniques for each individual query type don't change. Q/A site answers are still the best way to recommend for Q/A site questions, and bug report are still the most useful expertise source for assigning developers to bug reports.

While some expertise sources are only selectively useful, Tables 4.2 and 4.3 show that the code, commit, and all expertise sources have the best cross query performance and the highest average MRR. Whereas expertise sources such as Q/A site answers and feature requests do very poorly for most query types. This is because the commit message, code, and all expertise sources are sources that all developers have used and they demonstrate a large and varied amount of expertise.

Commits and commit messages occur with the same frequency. Whenever a developer makes a commit they add a commit message. But despite that commit messages surpass the MRR of code in four of the five query types. This could be because the vocabulary of source code is much larger and noisier than that of commit messages. Additionally commit messages take the changes made in commits and summarize them with natural language, which may be more likely to share vocabulary and semantics with the text in bug reports, Q/A site questions, or other query types.

In order to calculate each expert recommendation techniques cross query performance we average together the MRR's they obtain on the five different query types. This is also shown in Table 4.2. We observe that the top five expertise recommending

techniques when sorted by average MRR are:

1. All Sources and VSM
2. Commit Messages and VSM+LSI
3. All Sources and Naive Bayes
4. Commit Messages and VSM
5. All Sources and VSM+LSI

Using commit messages or all expertise sources at once to model expertise is the most consistently successful method across all five query types. Additionally VSM or VSM+LSI tend to perform the best.

We investigated if the rankings given by Commit Messages and VSM+LSI, All Sources and VSM, and Code and VSM (the best performing technique of the RQ1 techniques). we found that Commit Messages and VSM+LSI performs statistically significantly better than Code and VSM for Q/A site, bug report, feature request, and mailing list queries. All and VSM performed statistically significantly better than Code and VSM for Q/A site, bug report, and feature request queries. And All and VSM performed statistically significantly better than Commit Messages and VSM+LSI for feature request, mailing list, and code queries.

The combination of all expertise sources and Naive Bayes seems to be an outlier, with a noticeably higher average MRR than all other combinations of expertise sources and Naive Bayes. When a Naive Bayes classifier is trained on multiple expertise types it seems to be able to ignore the irrelevant data and obtain results relatively close to that of the best Naive Bayes classifier for a given query type.

We also observe an interesting distinction between the different query types when commit messages are used as the expertise source. Q/A site queries and mailing list queries have LSI outperform VSM. While bug report queries, feature request queries, and code queries have VSM outperform LSI.

Q/A site and mailing list short queries are questions written by OpenStack users and developers that want an answer from a developer with the relevant expertise. The bug report and feature request short queries aren't phrased as questions but are instead are short human written descriptions of broken functionality that needs to be fixed or new functionality that should be added. It could be that when people

write bug reports or feature requests that they use different semantics or vocabulary that work better with VSM, while the word choice of people asking questions on the Q/A site or mailing list could be different and cause LSI to be more useful. Additionally bug reports and feature requests are requests for expertise that involve directly editing the code and are likely to result in a commit message describing the change. If the past commit messages of developers mention and describe past bug reports or feature requests they worked on vocabulary similarity between messages and bug reports/feature requests could enhance VSM and make it more useful.

### **4.3.3 RQ3: What accuracy would existing expertise-finding techniques or our adapted expertise-finding techniques provide for short free-form expertise queries?**

RQ3 investigates how the accuracy of the expertise recommendation techniques change when we switch from using artifact specific queries that give the techniques an entire development artifact as input to short artifact agnostic queries. These short queries are generated from the original artifacts, either using only their title, summary, or a few distinct words as described in Section 4.2.6.

Table 4.4 shows the MRR values from the evaluations on the different query types, similar to Table 4.2. The same evaluation is done, but instead of using the full artifacts we used short queries.

We also display the MRR results as bar graphs in Figure 4.3.

The optimal expertise source for each individual query type is the same as when the full artifact was used, but the optimal expertise finding technique is different. The expertise recommendation techniques with the highest MRR for each individual expertise query type are:

- Q/A Site: Q/A Site Answers and Time TFIDF
- Bug Reports: Bug Reports and Time TFIDF
- Feature Requests: Feature Requests and VSM+LSI
- Mailing List: All Sources and Naive Bayes
- Code: Code and Time TFIDF

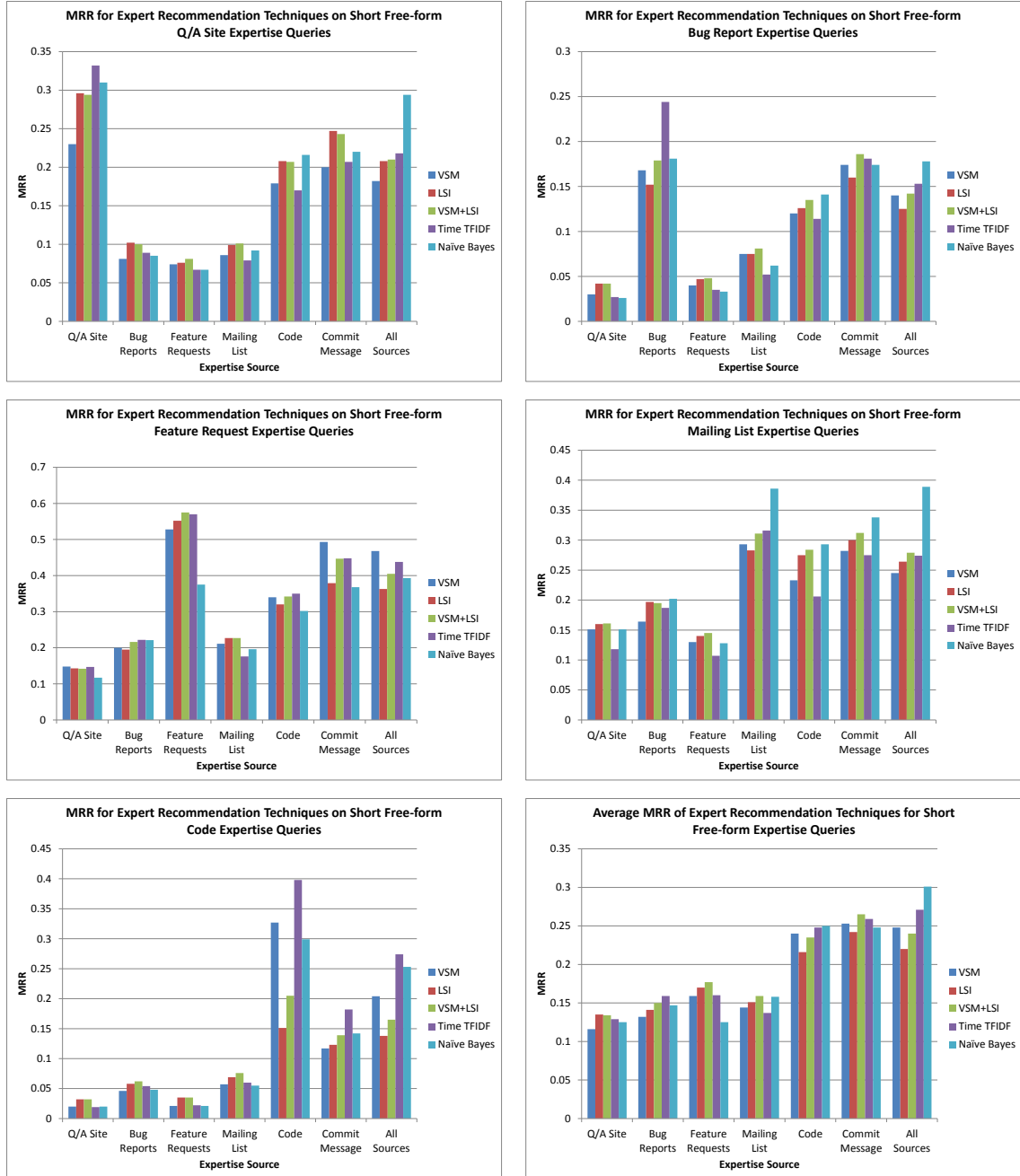


Figure 4.3: Results of RQ3 Evaluation



Table 4.4: MRR Results from evaluating expertise techniques using short free-form expertise queries

Expertise Source	Expertise Query	Naive Bayes	VSM	LSI	Time TFIDF	VSM+ LSI
Q/A Site	Q/A Site	0.310	0.230	0.296	0.332	0.294
Q/A Site	Bug Reports	0.085	0.081	0.102	0.089	0.100
Q/A Site	F. Requests	0.067	0.074	0.076	0.067	0.081
Q/A Site	Mailing List	0.092	0.086	0.099	0.079	0.101
Q/A Site	Code	0.216	0.179	0.208	0.170	0.207
Q/A Site	Commit Message	0.220	0.200	0.247	0.207	0.243
Q/A Site	All Sources	0.294	0.182	0.208	0.218	0.210
Bug Reports	Q/A Site	0.026	0.030	0.042	0.027	0.042
Bug Reports	Bug Reports	0.181	0.168	0.152	0.244	0.179
Bug Reports	F. Requests	0.033	0.040	0.047	0.035	0.048
Bug Reports	Mailing List	0.062	0.075	0.075	0.052	0.081
Bug Reports	Code	0.141	0.120	0.126	0.114	0.135
Bug Reports	Commit Message	0.174	0.174	0.160	0.181	0.186
Bug Reports	All Sources	0.178	0.140	0.125	0.153	0.142
F. Requests	Q/A Site	0.117	0.148	0.143	0.147	0.142
F. Requests	Bug Reports	0.221	0.200	0.195	0.222	0.216
F. Requests	F. Requests	0.375	0.528	0.552	0.570	0.575
F. Requests	Mailing List	0.196	0.211	0.227	0.176	0.227
F. Requests	Code	0.302	0.340	0.320	0.350	0.342
F. Requests	Commit Message	0.368	0.493	0.379	0.448	0.447
F. Requests	All Sources	0.393	0.468	0.363	0.438	0.405
Mailing List	Q/A Site	0.151	0.151	0.160	0.118	0.161
Mailing List	Bug Reports	0.202	0.164	0.197	0.187	0.195
Mailing List	F. Requests	0.128	0.130	0.140	0.107	0.145
Mailing List	Mailing List	0.386	0.293	0.283	0.316	0.311
Mailing List	Code	0.293	0.233	0.275	0.206	0.284
Mailing List	Commit Message	0.338	0.282	0.300	0.275	0.312
Mailing List	All Sources	0.389	0.245	0.264	0.274	0.279
Code	Q/A Site	0.020	0.020	0.032	0.019	0.032
Code	Bug Reports	0.048	0.046	0.058	0.054	0.062
Code	F. Requests	0.021	0.021	0.035	0.022	0.035
Code	Mailing List	0.055	0.057	0.069	0.060	0.076
Code	Code	0.299	0.327	0.151	0.398	0.205
Code	Commit Message	0.142	0.117	0.123	0.182	0.139
Code	All Sources	0.253	0.204	0.138	0.274	0.165
Average	Q/A Site	0.125	0.116	0.135	0.129	0.134
Average	Bug Reports	0.147	0.132	0.141	0.159	0.150
Average	F. Requests	0.125	0.159	0.170	0.160	0.177
Average	Mailing List	0.158	0.144	0.151	0.137	0.159
Average	Code	0.250	0.240	0.216	0.248	0.235
Average	Commit Message	0.248	0.253	0.242	0.259	0.265
Average	All Sources	0.301	0.248	0.220	0.271	0.240

The optimal expertise source for each query type hasn't changed, except for mailing list queries where All Sources is the best expertise source. Though there isn't a statistically significant difference between the rankings given by All Sources and Naive Bayes and the rankings given by Mailing List and Naive Bayes, and the two techniques have very close MRR, so they are equally useful for mailing list queries. It is still generally best to use the expertise source that comes from the same place as the queries when dealing with just a single query type. But now that we are using short expertise queries Time TFIDF is much more useful. It is the optimal ranking technique for three of the queries. This may be because Time TFIDF's fine grained time metadata utilization makes each individual word very important for determining the score for a developer. When large artifacts are used as queries it might include words that aren't useful for Time TFIDF or that negatively impact its performance. But when only a few words are used it may be more likely that these words are the important and useful ones that Time TFIDF works well with.

The short free-form expertise queries have the same problem where the best expertise source for one query type will typically do poorly for all other types. Q/A site answers still aren't using for recommending developers for feature requests and mailing list messages aren't the best way to model a developer's expertise with bug reports. Code, commit messages, and using all expertise sources at once are still the expertise source types with the best cross query results.

The expertise recommendation techniques with the top 5 average MRR across all 5 query types are:

1. All Sources and Naive Bayes
2. All Sources and Time TFIDF
3. Commit Messages and VSM+LSI
4. Commit Messages and Time TFIDF
5. Commit Messages and Naive Bayes

We investigated if the rankings given by Commit Messages and VSM+LSI, All Sources and Naive Bayes, and Code and Naive Bayes (the best performing technique of the RQ1 techniques). we found that both Commit Messages and VSM+LSI and All and Naive Bayes performed statistically significantly better than Code and Naive Bayes for Q/A site, bug report, feature request, and mailing list queries. And

All Sources and Naive Bayes performed statistically significantly better than Commit Messages and VSM+LSI for Q/A site and code queries.

Additionally we noticed that All and Naive Bayes and Commit Messages and VSM+LSI didn't have a statistically significant difference in their rankings for mailing list queries, even though there is a large difference in MRR's (0.389 and 0.312). On further inspection we found this is because All and Naive Bayes has a much higher Recall@1 than Commit Messages and VSM+LSI, but their Recall@5 are very close together, implying that the two techniques have similar Recall@X for high X's.

The combination of using all expertise sources at once and a Naive Bayes classifier obtains a noticeably higher MRR than the other techniques. This is because, similar to when full artifact queries were used, the Naive Bayes approach is able to take advantage of the training data coming from the optimal expertise source for a certain query while not being confused by irrelevant data from less useful expertise sources.

Time TFIDF is also useful when all expertise sources are used. This could be because while using all sources at once can introduce confusing or noisy data, Time TFIDF uses time metadata to focus on the words a developer has used the most recently. And developers are most likely to use words recently in expertise sources that they frequently use.

Commit messages are the best expertise source when only one type of source is used, and VSM+LSI is still the best expertise finding technique for commit messages even when short queries are used. But due to the loss of textual information Time TFIDF and Naive Bayes, when paired with commit messages, have a MRR very close to VSM+LSI.

This is because the switch from artifact specific to artifact agnostic queries affects the expertise finding techniques differently. The IR techniques, VSM, LSI, and VSM+LSI, have a large drop in MRR. Expertise recommendation techniques that use Naive Bayes also mostly drop in MRR but to a smaller extent. And a majority of expertise recommendation techniques that use Time TFIDF actually improve when short queries are used.

VSM, LSI, and VSM+LSI are able to effectively leverage the large amount of textual data offered by large artifact specific queries. Whereas Time TFIDF is best when given only a few specific words. Since Time TFIDF relies on fine grained time metadata, irrelevant or poorly chosen words could severely affect its accuracy. The IR techniques, such as VSM and LSI, are more resistant to irrelevant words. Naive Bayes tends to perform worse than the IR techniques, but it is less affected by the

switch to short queries and it is very useful for leveraging multiple types of expertise sources at once. Each expertise finding technique has its strengths and weaknesses.

## 4.4 Discussion

We performed an exhaustive evaluation of different combinations of expertise sources and expertise finding techniques across a diverse range of queries and found that the different sources and techniques had different strengths and weaknesses.

Expertise sources that cover a narrower range of expertise and are less frequently used, such as feature requests and Q/A site answers, are very useful for recommending developers for specific query types. But they do not do well for other queries created from different development artifacts. This could be because their vocabulary only covers a limited range of a developer's expertise. Additionally one major reason why expertise sources like Q/A site answers, feature requests, bug reports, and mailing list messages do so poorly is that not all developers create these artifacts. If only one type of expertise source is used then the expertise finding technique won't have any expertise data for the developers that never created artifacts of that specific type. This could be a benefit for a technique that focuses on recommending for a single task. If a technique only recommends developers for one type of task or artifact then only having expertise data on the developers that have previously worked with similar artifacts would be useful and the limited usage of the expertise source wouldn't be as detrimental. But an expertise source that is only used by a small subset of developers isn't useful for a general purpose expert developer recommendation tool that must work for many different types of expertise needs. Such a technique would need to have expertise information for all developers that covers a wide range of information. Techniques that focus only on developers that happen to use the Q/A site, or the feature request repository, have been shown to have poor accuracy on most query types.

The expertise sources with the best results when all five query types are considered are code, commit messages, and all sources. Commit messages typically outperform code even though they occur at the same rate. And using all sources at once normally outperforms using only commit messages.

When full artifacts are used as queries IR techniques such as VSM+LSI do best. But when only short artifact agnostic queries are used, which are meant to approximate the search queries users would actually use, IR techniques have a significant drop

in accuracy. Time TFIDF sees a large increase in MRR. Additionally combining all expertise sources with a Naive Bayes classifier obtains the best cross query MRR by a noticeable margin.

We also observed that different query types worked best with LSI while other query types worked best with VSM. Queries from different sources or made with different intent could have different semantics that lead to either VSM or LSI being more useful than the other.

While using all expertise sources with Naive Bayes obtains the best results with short queries it poses some difficulties. Combining many expertise sources together requires more maintenance and upkeep since the classifier needs to be trained on sources from many different repositories and areas. Additionally the classifier would need to be retrained often to keep it up to date.

If only one expertise source is to be used then commit messages would be the best. And while some projects may be hindered by a lack of properly descriptive commit messages, we have also found that using code as the expertise source outperforms the more specialized expertise sources as well.

The results of our evaluations show that previously existing expertise finding techniques can be adapted to obtain consistently good results for multiple types of queries. Techniques that utilize commit messages or all available expertise sources have consistent performance. While these techniques do not obtain the best performance for any one given query type, they have been shown to be the most useful expertise sources for recommending developers for diverse expertise queries. This is because the techniques that are specialized for a single query type do very well for one kind of query and poorly for all others. We found that using commit messages or all sources to train expertise finding techniques gives enough expertise data on enough developers to obtain consistently good results for many different query types.

## 4.5 Threats to Validity

### 4.5.1 Internal Validity

We assumed that the developers that answered each query were the only ones with relevant expertise to that query. That may not be the case. It could be that many developers knew how to fix a bug or how to answer a question and that the expertise

of the answering developer wasn't the sole reason they answered the query. The way that we get ground truth only gives us a narrow view of which developers have expertise with a given query. We only consider the developers that specifically fixed the bug report, or answered the Q/A site question, or work directly with a given query. While this assumption means we only know a small number of the developers that would have expertise with a given query, this is an assumption that has been commonly made in past bug triage and automatic expert developer recommendation research in the past [5], [55], [43].

We also assumed that the expertise queries generated from development artifacts were good stand-ins for queries that a user of general purpose expert recommendation tool would use. While they all are short and use natural language to describe an expertise need, they were not written for the intent of being use as a query for an expert search tool. Thus the queries used to create the datasets could be structured differently than those that would be written by developers if a general purpose expert recommendation tool was used in real life.

## 4.5.2 External Validity

The expertise query evaluations are only performed on one open source project, OpenStack. While a large number of queries are evaluated, the results might not be generalizable to other software projects.

# Chapter 5

## Conclusion

In this thesis we extend the existing literature on automatic expert developer recommendation by investigating how developer expertise can be modeled and measured in different ways and how expertise information can be used to address diverse development tasks. Past software development expertise finding research focused only on modeling positive expertise and only recommended developer for one kind of task or information need.

We first showed that modeling developer inexpertise is possible, that developer inexpertise is most prevalent in code changes where they make mistakes or introduce defects, and that developer inexpertise can be used to direct code inspection effort to help find incorrect or buggy code changes. We developed a metric for measuring developer expertise that can find which terms developers frequently misuse. Future work can look into modeling developer expertise in different ways, possibly using the vocabulary of the bugs they introduce or the questions they ask on Q/A sites. Modeling developer inexpertise in different ways could reveal new patterns or techniques that will make it easier to predict future mistakes.

We then evaluated a wide range of expert developer recommendation techniques to find the best technique for handling a diverse range of expertise queries. We also evaluated the performance of techniques using both long and short expertise queries. We found that most techniques did well on one kind of query but poorly on all others. And that most techniques did worse on short queries than long queries. Leveraging all available expertise sources with a Naive Bayes classifier obtained the best cross query results on short queries and commit messages with VSM+LSI got the best cross query results for a technique that only used one expertise source. Future work

could take the expertise finding techniques that obtained the best cross query results and implement a general purpose expert developer recommendation tool. This tool could then be used in a human study to evaluate how it handles a diverse array of human written search queries.

This work has shown how two different aspects of software developer expertise modeling can be looked at in more diverse ways and improved. These two studies can be used as jumping off points for future work to further improve how we model and utilize developer expertise. While neither of our studies found techniques that have accuracy far above past techniques, they look at developer expertise in different ways and can motivate future work. Future work on modeling developer inexpertise can focus on creating new ways to model inexpertise using different sources and creating algorithms that can leverage developer inexpertise to assist with many different development tasks. And future work on handling diverse expertise needs with a single technique can take the techniques that obtained the best cross query performance and evaluate them on search queries and ground truth obtained from surveying developers. Additionally a general purpose expert developer recommendation tool could be implemented and evaluated in an actual software development environment.



# Bibliography

- [1] Gerrit Code Review. <https://www.gerritcodereview.com>.
- [2] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, pages 216–221. IEEE, 2009.
- [3] Ibrahim Aljarah, Shadi Banitaan, Sameer Abufardeh, Wei Jin, and Saeed Salem. Selecting Discriminating Terms for Bug Assignment: a Formal Analysis. In *International Conference on Predictive Models in Software Engineering*, pages 12:1–12:7, 2011.
- [4] John Anvik and Gail C Murphy. Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions. *ACM Transactions On Software Engineering And Methodology*, 20(3):10:1–10:35, 2011.
- [5] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.
- [6] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.
- [7] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 931–940. IEEE, 2013.
- [8] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets.

- In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 134–144. IEEE Press, 2015.
- [9] O. Baysal, M.W. Godfrey, and R. Cohen. A bug you like: A framework for automated assignment of bugs. In *International Conference on Program Comprehension*, pages 297–298, 2009.
- [10] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.
- [11] Pamela Bhattacharya, Iulian Neamtiu, and Christian R. Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012.
- [12] Christopher S Campbell, Paul P Maglio, Alex Cozzi, and Byron Dom. Expertise identification using email communications. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 528–531. ACM, 2003.
- [13] Gerardo Canfora and Luigi Cerulo. How Software Repositories Can Help in Resolving a New Change Request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.
- [14] Gerardo Canfora and Luigi Cerulo. Supporting Change Request Assignment in Open Source Development. In *ACM Symposium on Applied Computing*, pages 1767–1772, 2006.
- [15] Yguaratã Cerqueira Cavalcanti, Paulo Anselmo Mota Silveira Neto, Ivan do Carmo Machado, Tassio Ferreira Vale, Eduardo Santana Almeida, and Silvio Romero de Lemos Meira. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process*, 26(7):620–653, 2014.
- [16] Davor Cubranic. Automatic bug triage using text categorization. In *International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.
- [17] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

- [18] G.A. Di Lucca, M. Di Penta, and S. Gradara. An Approach to Classify Software Maintenance Requests. In *International Conference on Software Maintenance*, pages 93–102, 2002.
- [19] Jinwen Guo, Shengliang Xu, Shenghua Bao, and Yong Yu. Tapping on the potential of q&a community by recommending answer providers. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 921–930. ACM, 2008.
- [20] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, AE Cruz, Kenji Fujiwara, and Hajimu Iida. Who does what during a code review? datasets of oss peer review repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 49–52. IEEE Press, 2013.
- [21] Md Kamal Hossen, Huzefa Kagdi, and Denys Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *International Conference on Program Comprehension*, pages 130–141. ACM, 2014.
- [22] Md Kamal Hossen, Huzefa Kagdi, and Denys Poshyvanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 130–141. ACM, 2014.
- [23] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Foundations of Software Engineering*, pages 111–120, 2009.
- [24] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.
- [25] W Lewis Johnson and Ali Erdem. Interactive explanation of software systems. *Automated Software Engineering*, 4(1):53–75, 1997.
- [26] H. Kagdi and D. Poshyvanyk. Who can help me with this change request? In *International Conference on Program Comprehension*, 2009.
- [27] Huzefa Kagdi and Denys Poshyvanyk. Who can help me with this change request? In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 273–277. IEEE, 2009.

- [28] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.
- [29] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2): 181–196, 2008.
- [30] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 344–353. IEEE, 2007.
- [31] Bruce Krulwich, Chad Burkey, and A Consulting. The contactfinder agent: Answering bulletin board questions with referrals. In *AAAI/IAAI, Vol. 1*, pages 10–15, 1996.
- [32] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.
- [33] Raula Gaikovina Kula, Ana E Carmago Cruz, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, Xin Yang, and Hajimu Iida. Using profiling metrics to categorise peer review types in the android project. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, pages 146–151. IEEE, 2012.
- [34] Thomas K Landauer and Susan T Dumais. A solution to plato’s problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211, 1997.
- [35] M. Linares-Vasquez, K. Hossen, Hoang Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *International Conference on Software Maintenance*, pages 451–460, 2012.
- [36] Mingrong Liu, Yicen Liu, and Qing Yang. Predicting best answerers for new questions in community question answering. In *International Conference on Web-Age Information Management*, pages 127–138. Springer, 2010.
- [37] David Ma, David Schuler, Thomas Zimmermann, and Jonathan Sillito. Expert recommendation with usage expertise. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 535–538. IEEE, 2009.

- [38] Mika V Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3): 430–448, 2009.
- [39] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *International Working Conference on Mining Software Repositories*, pages 131–140, 2009.
- [40] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 131–140. IEEE, 2009.
- [41] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240. ACM, 2000.
- [42] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th international conference on software engineering*, pages 503–512. ACM, 2002.
- [43] G Murphy and D Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004.
- [44] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. Bug report assignee recommendation using activity profiles. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 22–30. IEEE, 2013.
- [45] OpenStack. Software: Openstack open source cloud computing software, 2017. URL <https://www.openstack.org/software/>.
- [46] Jin-Woo Park, Mu-Woong Lee, Jinhan Kim, Seung-won Hwang, and Sunghun Kim. CosTriage: A Cost-Aware Triage Algorithm for Bug Reporting Systems. In *Conference on Artificial Intelligence*, pages 139–144, 2011.
- [47] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

- [48] Fatemeh Riahi, Zainab Zolaktaf, Mahdi Shafiei, and Evangelos Milios. Finding expert users in community question answering. In *Proceedings of the 21st International Conference on World Wide Web*, pages 791–798. ACM, 2012.
- [49] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.
- [50] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124. ACM, 2008.
- [51] Sam Scott and Stan Matwin. Feature engineering for text classification. In *ICML*, volume 99, pages 379–388, 1999.
- [52] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.
- [53] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.
- [54] Ramin Shokripour, John Anvik, Zarinah Mohd Kasirun, and Sima Zamani. Why so Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation. In *Mining of Software Repositories*, pages 2–11, 2013.
- [55] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. A time-based approach to automatic bug report assignment. *Journal of Systems and Software*, 102:109–122, 2015.
- [56] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34. ACM, 2006.
- [57] Jacek Sliwerski and Thomas Zimmermann Andreas Zeller. When do changes induce fixes? *Working Conference on Mining Software Repositories 2005*, 1 (1.18):1–5, 2005.

- [58] Kalyanasundaram Somasundaram and Gail C Murphy. Automatic categorization of bug reports using latent dirichlet allocation. In *Proceedings of the 5th India software engineering conference*, pages 125–130. ACM, 2012.
- [59] Ahmed Tamrawi, Tung T. Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Foundations of Software Engineering*, 2011.
- [60] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150. IEEE, 2015.
- [61] Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *Asia Pacific Software Engineering Conference*, pages 389–396. IEEE, 2011.
- [62] Xin Xia, Daniel Lo, Xinyu Wang, and Bo Zhou. Accurate developer recommendation for bug resolution. In *Working Conference on Reverse Engineering*, pages 72–81. IEEE, 2013.
- [63] Xin Xia, David Lo, Ying Ding, Jafar M Al-Kofahi, Tien N Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3):272–297, 2017.
- [64] Jifeng Xuan, He Jiang, Zhilei Ren, Jun Yan, and Zhongxuan Luo. Automatic bug triage using semi-supervised text classification. *arXiv preprint arXiv:1704.04769*, 2017.
- [65] Hui Yang, Xiaobing Sun, Yucong Duan, and Bin Li. On the effects of exploring historical commit messages for developer recommendation. *Chinese Journal of Electronics*, 25(4):658–664, 2016.
- [66] Xin Yang, Raula Gaikovina Kula, Camargo Cruz Ana Erika, Norihiro Yoshida, Kazuki Hamasaki, Kenji Fujiwara, and Hajimu Iida. Understanding oss peer review roles in peer review social network (person). In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 709–712. IEEE, 2012.

- [67] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.
- [68] Lotfi A Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.
- [69] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Using developer-interaction trails to triage change requests. In *Mining Software Repositories*, pages 88–98, 2015.



# Appendix A: Recall@1-5 for Chapter 4 Evaluations

Table 1: Recall@1-5 for Artifact Specific Q/A Site Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	17.78%	28%	37.09%	44.89%	50.68%
Q/A Site	LSI	24.05%	38.86%	49.72%	57.2%	62.91%
Q/A Site	VSM+LSI	23.09%	39.98%	50.36%	57.12%	62.19%
Q/A Site	Time TFIDF	16.89%	24.54%	32.5%	38.13%	43.52%
Q/A Site	Nave Bayes	18.83%	28.4%	34.59%	39.82%	43.04%
Bug Reports	VSM	5.95%	7.56%	9.25%	11.1%	12.15%
Bug Reports	LSI	6.76%	8.05%	10.14%	11.5%	12.55%
Bug Reports	VSM+LSI	6.6%	8.45%	10.46%	11.5%	12.39%
Bug Reports	Time TFIDF	4.26%	6.6%	7.56%	8.69%	9.73%
Bug Reports	Nave Bayes	4.99%	7%	8.77%	9.9%	10.54%
Feature Requests	VSM	3.7%	5.23%	6.76%	7.88%	8.69%
Feature Requests	LSI	3.54%	5.39%	6.36%	7.08%	7.32%
Feature Requests	VSM+LSI	3.7%	5.31%	6.28%	7.08%	7.8%
Feature Requests	Time TFIDF	1.45%	3.46%	4.67%	5.15%	6.68%
Feature Requests	Nave Bayes	3.38%	4.75%	6.36%	7.8%	8.93%
Mailing List	VSM	3.78%	7.32%	9.41%	11.99%	14.4%
Mailing List	LSI	4.51%	7.72%	10.38%	12.87%	15.37%
Mailing List	VSM+LSI	4.51%	7.96%	10.86%	13.35%	14.96%
Mailing List	Time TFIDF	2.25%	3.86%	4.51%	5.63%	6.68%
Mailing List	Nave Bayes	4.18%	6.92%	9.01%	11.02%	12.79%
Code	VSM	10.06%	18.26%	25.99%	30.89%	35.16%
Code	LSI	10.78%	19.47%	26.31%	30.97%	35.16%
Code	VSM+LSI	11.58%	20.27%	28%	31.62%	36.69%
Code	Time TFIDF	3.62%	7.48%	12.15%	15.29%	18.34%
Code	Nave Bayes	13.84%	21.4%	26.15%	31.3%	33.63%
Commit Message	VSM	11.99%	22.12%	29.53%	35.16%	38.7%
Commit Message	LSI	14.88%	26.31%	33.79%	40.95%	45.29%
Commit Message	VSM+LSI	15.77%	26.47%	33.79%	40.23%	46.26%
Commit Message	Time TFIDF	5.39%	9.98%	13.6%	16.25%	19.07%
Commit Message	Nave Bayes	13.27%	22.61%	27.84%	31.94%	34.92%
All Sources	VSM	9.49%	18.83%	27.11%	33.79%	39.5%
All Sources	LSI	12.71%	23.09%	32.5%	39.34%	45.13%
All Sources	VSM+LSI	12.39%	23.57%	33.31%	39.98%	46.58%
All Sources	Time TFIDF	5.23%	12.31%	17.06%	23.01%	26.87%
All Sources	Nave Bayes	20.76%	29.2%	35.56%	39.34%	42.4%

Table 2: Recall@1-5 for Artifact Specific Bug Report Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	1.75%	2.89%	3.72%	4.37%	4.94%
Q/A Site	LSI	1.92%	3.19%	3.91%	4.66%	5.35%
Q/A Site	VSM+LSI	1.96%	3.26%	3.99%	4.8%	5.4%
Q/A Site	Time TFIDF	0.77%	1.44%	1.95%	2.36%	2.8%
Q/A Site	Nave Bayes	1.55%	2.6%	3.54%	4.32%	5.01%
Bug Reports	VSM	13.03%	20.38%	25.28%	28.84%	31.75%
Bug Reports	LSI	12.15%	18.44%	23.89%	27.68%	31.17%
Bug Reports	VSM+LSI	14.78%	22.11%	27.38%	31.58%	35.08%
Bug Reports	Time TFIDF	12.77%	17.92%	21.05%	23.98%	26.02%
Bug Reports	Nave Bayes	15.1%	20.89%	24.68%	27.81%	30.29%
Feature Requests	VSM	2.53%	3.83%	4.75%	5.52%	6.12%
Feature Requests	LSI	2.45%	3.97%	4.93%	5.68%	6.2%
Feature Requests	VSM+LSI	2.5%	4.19%	5.06%	5.79%	6.39%
Feature Requests	Time TFIDF	1.04%	1.68%	2.12%	2.57%	2.99%
Feature Requests	Nave Bayes	1.92%	3.15%	4.02%	4.53%	5.14%
Mailing List	VSM	6.13%	9.04%	10.96%	12.33%	13.45%
Mailing List	LSI	5.38%	8.2%	10.19%	11.87%	13.19%
Mailing List	VSM+LSI	6.08%	9.15%	11.25%	12.75%	14.16%
Mailing List	Time TFIDF	1.37%	2.18%	2.88%	3.5%	4.08%
Mailing List	Nave Bayes	4.6%	6.78%	8.26%	9.62%	10.84%
Code	VSM	8.03%	13.75%	18.16%	21.99%	25.2%
Code	LSI	7.66%	13.25%	17.51%	20.94%	24.14%
Code	VSM+LSI	9.01%	14.72%	19.2%	22.65%	25.56%
Code	Time TFIDF	2.56%	4.76%	6.92%	8.53%	10.18%
Code	Nave Bayes	9.11%	13.87%	16.59%	18.7%	20.64%
Commit Message	VSM	13.22%	20.99%	26.81%	31.49%	35.27%
Commit Message	LSI	11.05%	17.87%	22.94%	27.21%	30.43%
Commit Message	VSM+LSI	14.11%	22.05%	27.72%	31.68%	35.34%
Commit Message	Time TFIDF	5.92%	8.78%	11.18%	13.09%	14.48%
Commit Message	Nave Bayes	13.38%	19.38%	23.19%	26.69%	29.5%
All Sources	VSM	11.96%	18.42%	22.77%	26.67%	29.94%
All Sources	LSI	8.57%	13.71%	18.04%	21.57%	24.92%
All Sources	VSM+LSI	11.16%	16.73%	22.06%	25.86%	28.98%
All Sources	Time TFIDF	5.37%	9.16%	11.96%	14.63%	16.76%
All Sources	Nave Bayes	13.81%	19.37%	23.05%	25.92%	28.3%

Table 3: Recall@1-5 for Artifact Specific Feature Request Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	9.22%	15.21%	18.43%	19.82%	21.66%
Q/A Site	LSI	8.29%	11.98%	17.05%	18.43%	18.43%
Q/A Site	VSM+LSI	8.29%	12.9%	17.05%	18.43%	19.35%
Q/A Site	Time TFIDF	5.99%	10.14%	13.36%	16.13%	17.97%
Q/A Site	Nave Bayes	9.68%	15.67%	18.43%	22.12%	22.58%
Bug Reports	VSM	11.98%	20.74%	26.73%	31.8%	36.87%
Bug Reports	LSI	15.21%	20.74%	27.19%	33.64%	38.25%
Bug Reports	VSM+LSI	14.29%	23.04%	26.73%	34.56%	37.33%
Bug Reports	Time TFIDF	11.06%	15.21%	17.51%	21.2%	22.58%
Bug Reports	Nave Bayes	17.97%	24.88%	28.57%	31.34%	35.48%
Feature Requests	VSM	52.07%	60.37%	64.06%	65.44%	67.28%
Feature Requests	LSI	51.15%	59.91%	62.67%	66.82%	69.12%
Feature Requests	VSM+LSI	53.92%	63.13%	65.44%	70.05%	70.97%
Feature Requests	Time TFIDF	49.77%	54.84%	58.53%	61.29%	64.06%
Feature Requests	Nave Bayes	35.94%	41.94%	45.16%	47%	49.31%
Mailing List	VSM	17.05%	22.58%	25.81%	28.57%	31.34%
Mailing List	LSI	16.13%	24.88%	29.03%	31.8%	35.02%
Mailing List	VSM+LSI	19.82%	27.65%	31.8%	33.64%	35.48%
Mailing List	Time TFIDF	5.99%	10.14%	13.36%	16.59%	19.35%
Mailing List	Nave Bayes	15.67%	23.5%	24.88%	29.03%	32.26%
Code	VSM	29.49%	41.94%	45.62%	51.15%	55.76%
Code	LSI	28.11%	40.55%	44.7%	50.69%	55.3%
Code	VSM+LSI	29.49%	40.55%	47.47%	50.23%	56.68%
Code	Time TFIDF	14.75%	21.66%	28.57%	32.26%	33.18%
Code	Nave Bayes	18.89%	31.34%	37.33%	40.55%	45.62%
Commit Message	VSM	46.54%	58.53%	62.21%	68.2%	70.05%
Commit Message	LSI	33.18%	45.16%	50.23%	55.3%	60.37%
Commit Message	VSM+LSI	40.55%	54.84%	58.53%	62.21%	65.9%
Commit Message	Time TFIDF	17.51%	21.66%	28.57%	29.95%	34.1%
Commit Message	Nave Bayes	29.03%	44.24%	51.15%	56.22%	58.06%
All Sources	VSM	48.85%	58.06%	69.12%	73.27%	74.65%
All Sources	LSI	32.26%	45.16%	50.23%	57.14%	60.37%
All Sources	VSM+LSI	39.17%	53%	54.84%	59.45%	63.13%
All Sources	Time TFIDF	24.42%	35.48%	43.78%	46.54%	49.77%
All Sources	Nave Bayes	32.72%	43.78%	51.15%	55.76%	57.14%

Table 4: Recall@1-5 for Artifact Specific Mailing List Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	6.91%	10.5%	13.12%	16.16%	17.96%
Q/A Site	LSI	7.73%	12.02%	15.06%	16.57%	17.96%
Q/A Site	VSM+LSI	7.32%	12.02%	15.61%	16.57%	17.54%
Q/A Site	Time TFIDF	3.04%	5.25%	7.04%	9.25%	11.46%
Q/A Site	Nave Bayes	7.73%	13.12%	15.88%	18.51%	20.72%
Bug Reports	VSM	9.39%	16.44%	22.1%	26.52%	29.83%
Bug Reports	LSI	9.94%	17.96%	22.79%	29.83%	33.15%
Bug Reports	VSM+LSI	10.36%	17.82%	24.59%	29.14%	33.7%
Bug Reports	Time TFIDF	7.46%	10.77%	14.5%	16.44%	18.51%
Bug Reports	Nave Bayes	9.94%	15.61%	18.92%	23.07%	25.97%
Feature Requests	VSM	6.49%	9.67%	11.74%	13.67%	14.78%
Feature Requests	LSI	6.35%	9.39%	12.29%	13.67%	15.47%
Feature Requests	VSM+LSI	6.77%	9.94%	12.43%	14.78%	16.16%
Feature Requests	Time TFIDF	3.59%	5.94%	8.84%	10.77%	11.74%
Feature Requests	Nave Bayes	4.83%	7.46%	10.36%	12.02%	13.81%
Mailing List	VSM	39.09%	45.58%	49.17%	53.31%	56.35%
Mailing List	LSI	26.24%	36.88%	46.13%	53.18%	56.77%
Mailing List	VSM+LSI	41.85%	51.1%	56.08%	60.36%	63.95%
Mailing List	Time TFIDF	21.27%	29.28%	36.33%	40.06%	42.13%
Mailing List	Nave Bayes	38.67%	45.99%	50.69%	54.01%	56.49%
Code	VSM	23.2%	35.5%	40.47%	43.09%	46.41%
Code	LSI	19.61%	29.97%	37.29%	42.13%	46.82%
Code	VSM+LSI	21.41%	33.7%	40.19%	44.34%	48.62%
Code	Time TFIDF	5.8%	11.19%	14.36%	18.51%	22.51%
Code	Nave Bayes	19.75%	27.9%	33.15%	36.33%	40.47%
Commit Message	VSM	20.58%	32.04%	40.61%	45.72%	49.86%
Commit Message	LSI	23.62%	34.94%	43.09%	49.31%	55.11%
Commit Message	VSM+LSI	24.86%	37.02%	45.86%	52.62%	57.04%
Commit Message	Time TFIDF	12.85%	18.51%	23.9%	28.73%	32.32%
Commit Message	Nave Bayes	19.75%	29.14%	35.08%	40.33%	44.2%
All Sources	VSM	33.43%	39.64%	43.23%	46.55%	50%
All Sources	LSI	19.75%	29.56%	36.33%	41.44%	44.75%
All Sources	VSM+LSI	27.76%	36.33%	42.82%	46.55%	51.24%
All Sources	Time TFIDF	11.05%	17.68%	24.31%	28.18%	32.6%
All Sources	Nave Bayes	32.32%	42.13%	48.76%	52.07%	54.83%

Table 5: Recall@1-5 for Artifact Specific Code Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	1.38%	2.19%	2.88%	3.48%	3.97%
Q/A Site	LSI	1.63%	2.68%	3.54%	4.17%	4.73%
Q/A Site	VSM+LSI	1.62%	2.69%	3.53%	4.14%	4.75%
Q/A Site	Time TFIDF	0.64%	1.09%	1.52%	1.92%	2.28%
Q/A Site	Nave Bayes	1.11%	1.89%	2.64%	3.22%	3.72%
Bug Reports	VSM	2.98%	4.91%	6.32%	7.39%	8.35%
Bug Reports	LSI	3.05%	5.17%	6.72%	8.1%	9.27%
Bug Reports	VSM+LSI	3.45%	5.61%	7.3%	8.7%	9.86%
Bug Reports	Time TFIDF	1.72%	2.83%	3.7%	4.43%	5.09%
Bug Reports	Nave Bayes	3%	4.6%	5.76%	6.73%	7.56%
Feature Requests	VSM	1.69%	2.6%	3.24%	3.64%	3.95%
Feature Requests	LSI	1.96%	3.04%	3.8%	4.41%	4.81%
Feature Requests	VSM+LSI	2.06%	3.14%	3.89%	4.45%	4.91%
Feature Requests	Time TFIDF	0.9%	1.51%	2.03%	2.38%	2.75%
Feature Requests	Nave Bayes	1.37%	2.19%	2.76%	3.19%	3.53%
Mailing List	VSM	4.85%	6.85%	8.33%	9.47%	10.49%
Mailing List	LSI	3.75%	5.85%	7.57%	9.01%	10.24%
Mailing List	VSM+LSI	4.45%	6.74%	8.58%	10.15%	11.38%
Mailing List	Time TFIDF	2.26%	3.49%	4.49%	5.33%	6.08%
Mailing List	Nave Bayes	4.11%	5.88%	7.09%	8.03%	8.89%
Commit Message	VSM	9.78%	15.46%	19.74%	22.9%	25.5%
Commit Message	LSI	9.5%	15.37%	19.49%	22.77%	25.58%
Commit Message	VSM+LSI	11.19%	17.66%	22.12%	25.76%	28.74%
Commit Message	Time TFIDF	8.1%	12.09%	14.77%	17.11%	19.14%
Commit Message	Nave Bayes	11.02%	16.35%	19.95%	22.63%	24.75%
Code	VSM	26.95%	36.04%	41.54%	45.51%	48.72%
Code	LSI	11.98%	17.74%	21.83%	25.11%	27.91%
Code	VSM+LSI	18.92%	25.75%	30.32%	34.07%	37.09%
Code	Time TFIDF	20.51%	27.66%	32.39%	35.84%	38.57%
Code	Nave Bayes	18.23%	25.32%	29.64%	32.72%	35.03%
All Sources	VSM	14.89%	22.23%	27.26%	31.19%	34.27%
All Sources	LSI	9.79%	14.82%	18.37%	21.3%	23.75%
All Sources	VSM+LSI	12.41%	18.4%	22.56%	25.75%	28.41%
All Sources	Time TFIDF	9.89%	15.69%	19.84%	22.99%	25.64%
All Sources	Nave Bayes	19.07%	26.19%	30.68%	33.9%	36.3%

Table 6: Recall@1-5 for Artifact Agnostic Q/A Site Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	8.37%	17.94%	26.15%	32.98%	39.42%
Q/A Site	LSI	14.24%	24.46%	32.66%	42.64%	49.72%
Q/A Site	VSM+LSI	14.24%	24.86%	33.79%	42.64%	48.91%
Q/A Site	Time TFIDF	19.95%	31.54%	38.46%	44.09%	48.35%
Q/A Site	Nave Bayes	18.83%	28.88%	35.24%	39.34%	43.77%
Bug Reports	VSM	4.34%	5.63%	7.56%	8.85%	10.3%
Bug Reports	LSI	6.11%	7.8%	9.17%	10.46%	11.18%
Bug Reports	VSM+LSI	5.63%	8.69%	10.14%	11.5%	12.47%
Bug Reports	Time TFIDF	4.75%	6.36%	7.24%	7.96%	9.01%
Bug Reports	Nave Bayes	3.86%	6.52%	8.05%	8.77%	9.98%
Feature Requests	VSM	3.46%	4.51%	5.79%	6.92%	7.4%
Feature Requests	LSI	3.86%	5.23%	5.95%	6.68%	7.32%
Feature Requests	VSM+LSI	4.26%	5.55%	6.6%	7.08%	8.13%
Feature Requests	Time TFIDF	2.25%	3.94%	5.07%	6.11%	7.16%
Feature Requests	Nave Bayes	2.49%	4.1%	5.23%	6.76%	7.4%
Mailing List	VSM	2.65%	5.63%	7.8%	9.73%	11.34%
Mailing List	LSI	4.18%	7.48%	10.14%	11.99%	14.8%
Mailing List	VSM+LSI	3.62%	7.96%	10.22%	12.79%	14.24%
Mailing List	Time TFIDF	3.86%	5.39%	6.84%	8.29%	9.17%
Mailing List	Nave Bayes	4.02%	7.08%	9.33%	10.78%	11.99%
Code	VSM	6.76%	12.87%	19.31%	23.49%	28.16%
Code	LSI	9.01%	16.81%	23.89%	28.96%	33.15%
Code	VSM+LSI	8.69%	16.73%	22.85%	28.72%	33.63%
Code	Time TFIDF	7.56%	13.03%	17.3%	23.01%	25.42%
Code	Nave Bayes	12.71%	19.15%	23.41%	25.99%	29.04%
Commit Message	VSM	8.13%	15.77%	22.77%	26.95%	30.57%
Commit Message	LSI	10.86%	19.87%	28.8%	35.88%	39.9%
Commit Message	VSM+LSI	9.81%	20.27%	28.96%	34.92%	40.06%
Commit Message	Time TFIDF	10.86%	18.18%	23.17%	26.95%	29.53%
Commit Message	Nave Bayes	12.55%	19.23%	23.65%	27.19%	30.01%
All Sources	VSM	6.03%	12.63%	19.47%	25.42%	29.44%
All Sources	LSI	7.88%	15.77%	23.09%	29.44%	34.67%
All Sources	VSM+LSI	7.48%	16.49%	23.81%	30.25%	34.92%
All Sources	Time TFIDF	10.06%	18.83%	25.1%	29.53%	34.03%
All Sources	Nave Bayes	19.95%	27.03%	32.5%	35.8%	38.94%

Table 7: Recall@1-5 for Artifact Agnostic Bug Report Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	1.02%	1.74%	2.36%	2.91%	3.42%
Q/A Site	LSI	1.57%	2.78%	3.6%	4.25%	4.75%
Q/A Site	VSM+LSI	1.6%	2.74%	3.6%	4.28%	4.77%
Q/A Site	Time TFIDF	0.8%	1.39%	1.89%	2.36%	2.84%
Q/A Site	Nave Bayes	0.95%	1.59%	2.13%	2.68%	3.07%
Bug Reports	VSM	7.43%	12.43%	15.73%	18.38%	20.53%
Bug Reports	LSI	6.52%	10.96%	14.53%	17.39%	20.14%
Bug Reports	VSM+LSI	8.38%	13.46%	17.48%	20.47%	23.27%
Bug Reports	Time TFIDF	13.9%	19.38%	23.38%	26.49%	28.89%
Bug Reports	Nave Bayes	11.38%	16.29%	19.52%	22.24%	24.46%
Feature Requests	VSM	1.69%	2.73%	3.49%	4.08%	4.6%
Feature Requests	LSI	1.9%	3.2%	4.22%	4.87%	5.54%
Feature Requests	VSM+LSI	1.95%	3.34%	4.24%	4.96%	5.59%
Feature Requests	Time TFIDF	1.41%	2.2%	2.82%	3.26%	3.76%
Feature Requests	Nave Bayes	1.42%	2.67%	3.43%	3.85%	4.2%
Mailing List	VSM	3.94%	5.96%	7.41%	8.58%	9.56%
Mailing List	LSI	3.51%	5.85%	7.57%	8.7%	10.05%
Mailing List	VSM+LSI	3.94%	6.34%	8.1%	9.5%	10.85%
Mailing List	Time TFIDF	2.3%	3.23%	4.09%	5.05%	5.82%
Mailing List	Nave Bayes	3.3%	4.98%	6.39%	7.45%	8.39%
Code	VSM	4.89%	8.88%	11.7%	14.24%	16.44%
Code	LSI	5.46%	9.3%	12.28%	14.53%	16.83%
Code	VSM+LSI	5.97%	10.17%	13.49%	16.25%	18.81%
Code	Time TFIDF	4.53%	8.02%	10.6%	13.09%	14.62%
Code	Nave Bayes	8.24%	12.44%	15.27%	17.24%	19.12%
Commit Message	VSM	8.85%	14.15%	18.02%	20.97%	23.49%
Commit Message	LSI	7.26%	12.6%	16.51%	19.51%	22.08%
Commit Message	VSM+LSI	9.26%	15.35%	19.46%	22.93%	25.31%
Commit Message	Time TFIDF	9.66%	14.28%	17.89%	20.28%	22.24%
Commit Message	Nave Bayes	10.35%	15.57%	19.11%	22.09%	24.27%
All Sources	VSM	6.97%	11.46%	14.86%	17.35%	20.1%
All Sources	LSI	6.05%	9.86%	12.82%	15.32%	17.45%
All Sources	VSM+LSI	7.05%	11.27%	15.03%	17.94%	20.2%
All Sources	Time TFIDF	8.15%	12.48%	16.15%	18.93%	21.65%
All Sources	Nave Bayes	11.15%	16.12%	19.54%	21.92%	23.99%



Table 8: Recall@1-5 for Artifact Agnostic Feature Request Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	8.29%	12.44%	13.82%	14.75%	16.59%
Q/A Site	LSI	5.99%	11.06%	11.98%	16.13%	18.43%
Q/A Site	VSM+LSI	5.53%	10.6%	13.82%	16.13%	20.28%
Q/A Site	Time TFIDF	7.83%	10.6%	12.44%	15.21%	17.05%
Q/A Site	Nave Bayes	5.53%	9.22%	9.22%	11.98%	14.75%
Bug Reports	VSM	7.83%	17.97%	23.5%	28.57%	32.26%
Bug Reports	LSI	8.29%	15.21%	21.2%	27.19%	32.26%
Bug Reports	VSM+LSI	9.22%	17.51%	23.04%	29.95%	35.48%
Bug Reports	Time TFIDF	12.44%	17.05%	20.28%	24.88%	27.19%
Bug Reports	Nave Bayes	12.44%	17.05%	23.04%	29.95%	33.64%
Feature Requests	VSM	41.01%	50.69%	55.76%	60.37%	63.13%
Feature Requests	LSI	45.16%	53.46%	58.99%	62.21%	64.06%
Feature Requests	VSM+LSI	46.54%	58.99%	61.29%	63.59%	67.74%
Feature Requests	Time TFIDF	49.77%	56.68%	60.37%	64.52%	65.44%
Feature Requests	Nave Bayes	27.65%	37.33%	42.86%	44.7%	47%
Mailing List	VSM	11.52%	17.51%	19.35%	23.5%	25.81%
Mailing List	LSI	12.9%	18.89%	21.66%	27.19%	33.64%
Mailing List	VSM+LSI	12.44%	19.82%	25.35%	28.57%	32.26%
Mailing List	Time TFIDF	9.68%	17.51%	20.28%	21.66%	24.42%
Mailing List	Nave Bayes	11.98%	17.97%	20.28%	22.12%	24.42%
Code	VSM	20.28%	29.49%	35.94%	39.17%	41.94%
Code	LSI	19.35%	31.34%	36.87%	41.47%	45.16%
Code	VSM+LSI	21.66%	31.8%	38.25%	41.94%	46.08%
Code	Time TFIDF	18.89%	31.34%	39.17%	42.4%	45.62%
Code	Nave Bayes	19.35%	26.73%	33.18%	40.09%	43.78%
Commit Message	VSM	39.17%	47%	51.61%	54.38%	57.6%
Commit Message	LSI	23.96%	35.94%	43.32%	45.62%	52.53%
Commit Message	VSM+LSI	30.88%	42.86%	48.39%	52.53%	57.14%
Commit Message	Time TFIDF	29.95%	39.63%	47.93%	51.15%	53.92%
Commit Message	Nave Bayes	24.42%	36.87%	43.78%	46.08%	49.31%
All Sources	VSM	35.48%	45.62%	48.85%	57.6%	60.83%
All Sources	LSI	23.96%	35.02%	40.09%	43.32%	47.93%
All Sources	VSM+LSI	26.27%	41.01%	44.7%	49.77%	53.46%
All Sources	Time TFIDF	30.41%	41.94%	50.23%	52.53%	55.76%
All Sources	Nave Bayes	27.19%	39.63%	46.54%	49.31%	51.61%

Table 9: Recall@1-5 for Artifact Agnostic Mailing List Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	8.15%	12.43%	16.02%	18.09%	19.61%
Q/A Site	LSI	8.98%	14.36%	17.54%	18.78%	20.44%
Q/A Site	VSM+LSI	9.12%	13.4%	17.4%	19.48%	21.69%
Q/A Site	Time TFIDF	6.35%	9.25%	10.64%	13.67%	16.44%
Q/A Site	Nave Bayes	8.43%	13.67%	15.88%	17.13%	17.82%
Bug Reports	VSM	6.22%	12.29%	15.75%	19.75%	24.45%
Bug Reports	LSI	9.53%	13.67%	21.82%	26.38%	30.66%
Bug Reports	VSM+LSI	8.29%	14.5%	22.1%	26.1%	30.11%
Bug Reports	Time TFIDF	8.98%	12.71%	18.37%	22.24%	24.86%
Bug Reports	Nave Bayes	10.77%	16.57%	19.48%	24.31%	27.62%
Feature Requests	VSM	7.18%	12.02%	13.81%	15.19%	16.71%
Feature Requests	LSI	7.46%	12.98%	16.16%	19.2%	20.58%
Feature Requests	VSM+LSI	7.6%	13.54%	16.16%	19.06%	20.17%
Feature Requests	Time TFIDF	5.11%	8.56%	10.77%	12.71%	14.36%
Feature Requests	Nave Bayes	7.04%	10.36%	12.71%	13.67%	15.61%
Mailing List	VSM	16.16%	26.8%	34.25%	40.19%	42.96%
Mailing List	LSI	15.19%	26.1%	34.25%	39.92%	44.34%
Mailing List	VSM+LSI	17.96%	28.87%	37.57%	44.06%	47.79%
Mailing List	Time TFIDF	20.3%	29.83%	36.6%	42.82%	46.82%
Mailing List	Nave Bayes	25.69%	36.46%	43.92%	48.76%	52.62%
Code	VSM	11.74%	19.34%	25.14%	29.97%	33.7%
Code	LSI	15.19%	26.52%	33.56%	38.26%	42.4%
Code	VSM+LSI	16.02%	26.52%	33.56%	37.43%	42.82%
Code	Time TFIDF	7.6%	16.57%	22.24%	27.9%	34.25%
Code	Nave Bayes	17.96%	24.03%	30.39%	35.36%	40.19%
Commit Message	VSM	16.99%	25.97%	30.52%	35.22%	39.09%
Commit Message	LSI	16.3%	27.35%	34.94%	41.85%	47.79%
Commit Message	VSM+LSI	17.13%	28.31%	34.94%	43.23%	47.65%
Commit Message	Time TFIDF	15.88%	24.31%	29.83%	35.08%	39.09%
Commit Message	Nave Bayes	20.72%	31.35%	39.36%	44.75%	49.59%
All Sources	VSM	12.43%	20.3%	27.62%	33.7%	38.12%
All Sources	LSI	14.5%	25.69%	31.91%	37.15%	40.75%
All Sources	VSM+LSI	15.06%	27.21%	33.43%	38.4%	44.06%
All Sources	Time TFIDF	14.36%	23.48%	31.35%	35.36%	41.57%
All Sources	Nave Bayes	24.45%	36.05%	43.09%	48.48%	50.83%

Table 10: Recall@1-5 for Artifact Agnostic Code Queries

Expertise Source	Technique	Recall@1	Recall@2	Recall@3	Recall@4	Recall@5
Q/A Site	VSM	0.9%	1.42%	1.8%	2.13%	2.4%
Q/A Site	LSI	1.38%	2.39%	3.15%	3.75%	4.32%
Q/A Site	VSM+LSI	1.36%	2.41%	3.16%	3.75%	4.34%
Q/A Site	Time TFIDF	0.88%	1.4%	1.77%	2.08%	2.35%
Q/A Site	Nave Bayes	0.69%	1.18%	1.57%	1.89%	2.21%
Bug Reports	VSM	2.08%	3.5%	4.48%	5.26%	6.03%
Bug Reports	LSI	2.28%	4.11%	5.56%	6.8%	7.99%
Bug Reports	VSM+LSI	2.59%	4.47%	6.01%	7.23%	8.36%
Bug Reports	Time TFIDF	2.82%	4.35%	5.51%	6.38%	7.13%
Bug Reports	Nave Bayes	2.34%	3.71%	4.73%	5.57%	6.27%
Feature Requests	VSM	1.11%	1.71%	2.09%	2.32%	2.52%
Feature Requests	LSI	1.75%	2.82%	3.67%	4.21%	4.67%
Feature Requests	VSM+LSI	1.8%	2.86%	3.7%	4.29%	4.72%
Feature Requests	Time TFIDF	1.18%	1.72%	2.09%	2.35%	2.55%
Feature Requests	Nave Bayes	0.92%	1.48%	1.89%	2.19%	2.45%
Mailing List	VSM	3.28%	4.62%	5.62%	6.46%	7.15%
Mailing List	LSI	3.27%	5.18%	6.9%	8.24%	9.35%
Mailing List	VSM+LSI	3.89%	5.93%	7.65%	9.04%	10.13%
Mailing List	Time TFIDF	3.43%	4.91%	5.97%	6.84%	7.46%
Mailing List	Nave Bayes	3.17%	4.49%	5.48%	6.22%	6.91%
Code	VSM	22.21%	31.65%	37.31%	41.3%	44.35%
Code	LSI	7.31%	11.93%	15.4%	18.58%	21.23%
Code	VSM+LSI	10.98%	17.61%	22.21%	26.23%	29.55%
Code	Time TFIDF	30.24%	39.32%	44.48%	48.19%	50.75%
Code	Nave Bayes	16.88%	23.57%	27.75%	30.76%	33.13%
Commit Message	VSM	5.94%	9.81%	12.71%	15.04%	17.06%
Commit Message	LSI	5.36%	9.56%	12.84%	15.52%	17.93%
Commit Message	VSM+LSI	6.48%	11.29%	14.79%	17.76%	20.25%
Commit Message	Time TFIDF	11.82%	17.24%	20.67%	23.09%	25.02%
Commit Message	Nave Bayes	8.47%	12.71%	15.51%	17.76%	19.46%
All Sources	VSM	11.62%	18.05%	22.74%	26.32%	29.23%
All Sources	LSI	6.78%	11.05%	14.31%	17.03%	19.43%
All Sources	VSM+LSI	8.69%	13.68%	17.55%	20.59%	23.18%
All Sources	Time TFIDF	18.08%	25.86%	30.64%	34.15%	36.94%
All Sources	Nave Bayes	17.25%	24.13%	28.35%	31.5%	33.87%