RECOMMENDING EXPERT DEVELOPERS USING USAGE AND IMPLEMENTATION EXPERTISE

SHARMIN AKTER
Bachelor of Science (Engineering), Mawlana Bhashani Science and Technology University, 2014

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Sharmin Akter, 2021

RECOMMENDING EXPERT DEVELOPERS USING USAGE AND IMPLEMENTATION EXPERTISE

SHARMIN AKTER

Date of Defence: December 18, 2020

Dr. John Anvik Thesis Supervisor	Associate Professor	Ph.D.
Dr. Wendy Osborn Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. John Zhang Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Yllias Chali Chair, Thesis Examination Committee	Professor	Ph.D.

Dedication

This thesis is dedicated to my loving family members, who always believed in me.

Especially, my beloved parents – *Jharna Begum* and *Md Mahbubur Rahman*,

my first mentor and eldest sister – *Mahmuda Rahman Shema*,

and my dearest husband – *Mir Md Azizur Rahman Faiem*.

Thank you for supporting me in every stage of life, always encouraging me to achieve my goals, teaching me to dream big, and, most importantly, being my strength.

Abstract

Knowing the expert developers of a software development project has great significance in large-scale and geographically distributed projects. However, finding these expert developers can be challenging, which becomes more complicated over time as the development team gets bigger and more distributed. This thesis presents an expert developer recommender system for methods, based on the *usage expertise*, *implementation expertise*, and the *combination* of both, for the developers of a software project. A developer acquires usage expertise on a method by calling or using it and implementation expertise by creating or modifying it. To determine the method expertise of the developers, we mine both the source code repository and the version histories of a software development project. We determine the accuracy of our system by calculating the percentage of successful recommendations within the Top-N results. Through several experiments, we found that our recommender system produces around 90% average accuracy for Top-10 recommendations.

Acknowledgments

First and foremost, I would like to thank Almighty ALLAH for showering blessings upon me, without which this research would not have been possible.

I would like to express my deepest gratitude to my thesis supervisor Dr. John Anvik, for his immense support, sincere guidance, and endless encouragement, throughout my graduate experience. I have learned everything about research work under his proficient supervision. He was always there whenever I needed any help, and for him, my graduate experience was so smooth.

I am very grateful to my M.Sc. supervisory committee members Dr. Wendy Osborn and Dr. John Zhang, for their valuable feedback and input of time. I want to extend my thanks to my amazing Sibyl Lab members for always being encouraging.

I am extremely grateful to my parents, for their love, prayers, and sacrifices and for preparing me for my future. I want to thank and remember my sisters Shema, Rimu (deceased), and Tania, my brother Rafsan, my precious nieces Joa, Fayha, and Fayona, and adored nephew Taaj for their emotional support during my stressful times. I would also like to thank my in-laws for always being supportive and my husband for being my motivation.

Finally, I acknowledge the funding provided by the Natural Science and Engineering Research Council of Canada (NSERC) for this research. I am grateful to NSERC for funding this work.

Contents

Contents vi

List of Tables viii

List of Figures ix

1 Introduction 1

2 Background and Related Work 7
- 2.1 Background . . . 7
  - 2.1.1 Conceptualizing Knowledge . . . 7
  - 2.1.2 Changesets . . . 8
- 2.2 Related Work . . . 9
  - 2.2.1 Usage Expertise Recommenders . . . 10
  - 2.2.2 Implementation Expertise Recommenders . . . 11
  - 2.2.3 Domain Specific Expertise Recommenders . . . 12
- 2.3 Summary . . . 13

3 Expertise Recommender Creation Methodology 14
- 3.1 Expertise Definitions . . . 14
  - 3.1.1 Usage Expertise . . . 15
  - 3.1.2 Implementation Expertise . . . 17
- 3.2 Recommender Creation Approach . . . 19
  - 3.2.1 Mining the Version Control Repository . . . 20
  - 3.2.2 Mining the Source Code Repository . . . 21
  - 3.2.3 Calculating Expertise Scores . . . 22
  - 3.2.4 Determining Overall Expertise . . . 23
- 3.3 Recommending Experts . . . 24
- 3.4 Summary . . . 25

4 Evaluation Methodology 26
- 4.1 Data Collection . . . 27
- 4.2 Evaluation Procedure . . . 28
  - 4.2.1 Training the Recommender . . . 29
  - 4.2.2 Gathering Expertise Data . . . 33
  - 4.2.3 Generating Recommendations . . . 33
  - 4.2.4 Comparing Recommendations . . . 37
  - 4.2.5 Investigating Unsuccessful Recommendations . . . 38

4.2.6 Computing Evaluation Metrics 39
4.3 Summary 41

5 Results and Discussion 42
5.1 Results 42
5.1.1 Accuracy 43
5.1.2 Mean Reciprocal Rank (MRR) 51
5.2 Comparison to Previous Approach 54
5.2.1 Depth of Method Knowledge 54
5.2.2 Method Change Frequency 54
5.2.3 Performance Analysis 55
5.3 Threats to Validity 57
5.3.1 Construct Validity 57
5.3.2 External Validity 57
5.3.3 Internal Validity 58
5.4 Summary 61

6 Conclusion and Future Work 62
6.1 Future Work 64
6.1.1 Cleanup repetitive data 64
6.1.2 Filter comments and blank lines 64
6.1.3 Work with deletions 65
6.1.4 Add support for other programming languages 65
6.1.5 Categorize methods and developer skills 65
6.1.6 Experiment with more heuristics 65

Bibliography 66

Appendix A Recommendation Results 68

Appendix B Expertise Accuracy Graphs 71
B.1 Google's Guava Graphs 72
B.1.1 Usage Expertise Graphs 72
B.1.2 Implementation Expertise Graphs 73
B.1.3 Combined Expertise Graphs 74
B.2 Apache Commons Collections Graphs 79
B.2.1 Usage Expertise Graphs 79
B.2.2 Implementation Expertise Graphs 80
B.2.3 Combined Expertise Graphs 81
B.3 Eclipse's AspectJ Graphs 86
B.3.1 Usage Expertise Graphs 86
B.3.2 Implementation Expertise Graphs 87
B.3.3 Combined Expertise Graphs 88

List of Tables

4.1 Details of the Software Project Datasets (as of July 2020) 28
4.2 Weights assigned on the Usage and Implementation expertise scores 35

5.1 Average Method-wise Accuracy Results (Top-1) 43
5.2 Average Method-wise Accuracy Results (Top-5) 44
5.3 Average Method-wise Accuracy Results (Top-10) 44
5.4 Average Method-wise Accuracy Results Across All Projects 45
5.5 Average Commit-wise Accuracy Results (Top-1) 48
5.6 Average Commit-wise Accuracy Results (Top-5) 48
5.7 Average Commit-wise Accuracy Results (Top-10) 48
5.8 Average Commit-wise Accuracy Results Across All Projects 49

List of Figures

2.1 Categorizing methods based on a developer's levels of knowledge and types of expertise . . . 7
2.2 A change diff snap from the *Apache Commons Collections* project . . . 9

3.1 Changeset snapshot from commit #484 of project *AspectJ* . . . 16
3.2 The Recommender System Workflow . . . 19
3.3 Recommendations for method `hasNext` from project `commons-collections` 24

4.1 Method with nested method declaration and invocations . . . 32
4.2 Usage and Implementation Expertise Recommendations for method `keySet` from commit #3294 of project `guava` . . . 34
4.3 Overall Expertise Recommendations for method `keySet` from commit #3294 of project `guava` . . . 36

5.1 Average accuracy graph of Usage Expertise data for Top-10 (*AspectJ*) . . . 50
5.2 MRR chart with Implementation Expertise data for project *Guava* . . . 51
5.3 Method-wise MRR graphs with Combined Expertise data at WeightSet-3 (`0.5, 0.5`) for projects (a) *Guava*, (b) *Commons-Collections*, and (c) *AspectJ* . . . 53
5.4 Performance analysis graphs for both of the approaches for project *AspectJ* 55

A.1 Recommendation results for method overloading . . . 68
A.2 Recommendation results when usage expertise not found . . . 69
A.3 Recommendation results when implementation expertise not found . . . 70

B.1 Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 72
B.2 Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 73
B.3 Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 74
B.4 Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 75
B.5 Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 76
B.6 Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 77
B.7 Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *Guava* . . . 78

B.8 Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 79
B.9 Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* . . . 80
B.10 Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 81
B.11 Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 82
B.12 Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 83
B.13 Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 84
B.14 Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections* 85
B.15 Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 86
B.16 Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 87
B.17 Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 88
B.18 Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 89
B.19 Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 90
B.20 Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 91
B.21 Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ* 92

Chapter 1

Introduction

In software industries, the number of large-scale and geographically distributed software development projects is increasing every day. The decentralized and distributed nature of software industries has triggered the necessity of effective communication and coordination among the developers [10]. Whenever a developer joins a team and starts working on a project, most of the time s/he does not know much about either the project architecture or the specific modules, depending on their prior experience with the system. As a consequence, developers can find it challenging to choose the correct method required to accomplish the task assigned to them. This can also occur when a developer is assigned a feature that requires the use of unfamiliar projects or external libraries. At such points, they feel the necessity of discussing with or taking suggestions from an expert who has used or called a method frequently [6].

Moreover, a software component has a higher chance of failure if it has many *minor contributors* [4]. Here, minor contributor of a software project component refers to a developer who makes less than a 5% contribution to that component. Having knowledge of developers who are experts of a method could help the minor contributors to prevent faults and write bug-free code. Additionally, in many software projects, the developers do not have enough time to make proper documentation for their project. Even if some developers manage to create documentation, it generally is not complete, is outdated, or is not helpful enough to understand the usage of each method of a project. Obtaining instructions from an expert in such circumstances could prevent mistakes.

Lastly, the team manager of a project also needs to know about the expert developers to utilize the available resources in the most efficient way. Many organizations determine experts manually, which is neither entirely accurate nor always up-to-date. Some use automatic procedures, but those may not be appropriate for new developers as they typically have generated little data for use by these procedures. The approaches can be overly general, as well. Although these approaches may work for smaller development teams, they often fail to bring results for large-scale and geographically distributed teams [6, 10].

To address the problems discussed above and as a solution to the complex problem of finding expert developers, we created an expert recommender system based on the *usage expertise* and the *implementation expertise* of software project developers[1]. The concept of usage expertise was first introduced by Schuler and Zimmermann [10]. It refers to the expertise a developer accumulates on a method of a project or a library method by calling or using it within the code they write. In other words, a developer with usage expertise on a method knows how to use that method or how to call it and what purpose that method serves. On the other hand, a developer accumulates implementation expertise on a method of a project by writing it (i.e., by creating it or modifying it). The more a developer works on or uses a method, the more expertise the developer gains for that method. New developers and minor contributors can benefit from these recommendations, as it may save a significant amount of development time that is wasted through misunderstandings and thereby increase productivity.

In this thesis, we compare the work and results of Schuler and Zimmermann [10] and Ma et al. [7] against our approach. Ma et al. [7] extended the work of Schuler and Zimmerman [10] by assessing the validity of their concept. They investigated both of the usage expertise and implementation expertise models. In our approach, we explore a unified recommender and model of expertise for the developers of a project, along with the individual expertise models. We present a ranked list of expert developers for the methods of a project

[1]There can be cases where a developer can have only usage expertise or only implementation expertise for a method(see Subsection 3.2.4).

where the ranking is done based on the overall expertise — a combination of usage and implementation expertise of the developers. We also generate the ranked lists for the methods with the individual expertise models. With the individual usage expertise model, we can also recommend expert developers for the external API (Application Programming Interface) or library methods. In addition, our approach resolves method signature types, which Ma et al. [7] did not do.

To build the expertise recommender, we mine both the source code repository and the version histories of a software project to extract the usage and implementation contribution of each developer working on it. From the changesets in a version control system, we detect the method calls made and the methods added or modified by each developer throughout the project. To determine the origin of the methods used and implemented by the developers, we mine the source code repository of the dataset project while working with each commit. We use frequency mining to extract the usage instances and implementation contributions of the developers for the methods of a project. After finding the contribution data, we combine the values for the two types of expertise to determine the developers' overall method expertise. To distinguish the impact of the usage and implementation expertise on the developers' overall expertise, we assign weights ranging from 0 to 1 to the raw usage and implementation expertise scores of the developers while calculating the overall expertise. A user can search for the experts of any method of a project or an API by entering the method's name, its class name, and its package name in our system. The system provides a ranked list of expert developers instead of only the one who has worked on the method the most. This list offers additional options to look for in case of unavailability of the most expert developer.

The goal of this work is to seek the answers to the following research questions:

RQ1: Does our combined expertise model outperform our individual expertise models?

RQ2: Does the project and development team size affect the overall results?

RQ3: What combination of usage and implementation expertise produces better results?

RQ4: Do our individual and combined expertise models outperform the previous approach by Ma et al. [7]?

In our evaluation strategy, we determine the accuracy of our system by calculating the percentage of successful recommendations within the Top-N results. A recommendation is stated successful when the actual expert developer, who is the actual commit author of that method usage or implementation, is listed within the Top-N recommendation results [7]. Ma et al. [7] evaluated their recommender system using the projects *AspectJ* and *Eclipse*, where *AspectJ* is a plugin or tool for *Eclipse*. We have used three software dataset projects of different sizes and from different organizations to validate the system. Our chosen projects are *Google's Guava*, *Apache Commons Collections*, and *Eclipse's AspectJ*. In order to compare our results with that of Ma et al.'s [7], we have experimented with one common dataset project.

We computed average accuracy of our system and Mean Reciprocal Rank (MRR) for the commits using the results generated by the recommender for these dataset projects. We explored the average accuracy of our system for both of the individual expertise models and the combined expertise model for Top-1, Top-5, and Top-10 recommendations. Generalizing the results for the models across all of the dataset projects, we found that we achieved accurate recommendations for more than 50% of the query methods on average when we considered accuracy for Top-1. The percentage increased to 66%-79% when we considered accuracy for Top-5, and to almost the same average percentage (66%-81%) when we looked at accuracy for Top-10. These accuracy results reflect the mean percentage of query methods, for which successful recommendations are made. Furthermore, we computed the average commit-wise accuracy results, reflecting the percentage of query commits for which successful recommendations are made. The generalized results of our commit-wise operations show that we achieved accurate recommendations for 67%-75% of the query commits on average when we considered accuracy for Top-1. The percentage increased to

`78%-91%` when we considered accuracy for Top-5, and to almost the same average percentage (`78%-92%`) when we looked at accuracy for Top-10. Analyzing our generalized results we found that regardless of the number of recommended expert developers, the individual expertise models always perform better than the combined expertise model. Of the individual expertise models, the usage expertise models outperforms the implementation expertise models in most cases. Looking at the accuracy results project-wise, we found that *AspectJ* provides the highest accuracy among all three dataset projects for both of the individual and the combined expertise models, regardless of the number of recommended expert developers. For representing our recommendation results over time, we plotted graphs with the results obtained for the dataset projects. We observed a similar pattern in all the graphs for all of the dataset projects. We computed the MRR and plotted graphs with the generated results for the query commits of our dataset projects. However, we found that neither the results nor the charts provided meaningful data to evaluate the recommender system. Therefore, we investigated a method-based MRR for the Top-15 most frequently used and implemented methods of each dataset project. When we analyzed the results and plotted bar graphs with the results, we found that the MRR findings resemble our accuracy findings.

To compare our results with that of Ma et al. [7], we examine the Depth of Method Knowledge (equivalent to Usage Expertise) and the Method Change Frequency (equivalent to Implementation Expertise) heuristics. We used the results for project *AspectJ* from both of the approaches, as this is the only common dataset project between the two approaches. After comparing the results for both of the heuristics, we found that our recommender creation approach outperforms that of Ma et al. [7] including our combined expertise model, which represents a combination of Ma et al.'s [7] usage expertise and implementation expertise models.

We have organized our thesis in the following manner. First, we discuss some background information about our approach to creating expert developer recommender system in Chapter 2. This chapter also presents some discussions about the previous approaches

to create recommender system using usage expertise, implementation expertise, and general or domain specific expertise. Next in Chapter 3, we discuss the complete procedure of our recommender creation approach, where Chapter 4 explains our evaluation methodology in detail. The results of our experiments and the findings of this study are presented in Chapter 5. This chapter also includes discussion about our research questions and the potential threats to the validity of our results. Finally, Chapter 6 concludes this thesis with a discussion about the future directions of this work.

Chapter 2

Background and Related Work

This chapter discusses some background information to understand the expert developer recommendation system creation process proposed in this work. It also presents the previous research work that is complementary to our approach.

2.1 Background

2.1.1 Conceptualizing Knowledge

Based on the types of expertise and their levels, we can categorize and illustrate the knowledge of developers on methods, as presented in Figure 2.1. The illustration is inspired by the one proposed by Ye and Fischer [18] to depict different knowledge levels about high-functionality application (HFA).

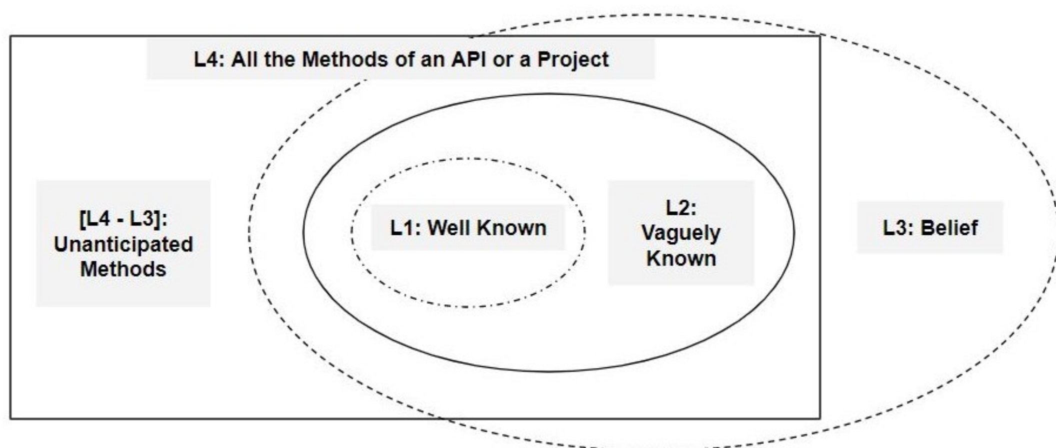


Figure 2.1: Categorizing methods based on a developer's levels of knowledge and types of expertise

In Figure 2.1, the rectangle L4 represents all the methods of a project or an API library. The ovals L1, L2, and L3 represent a particular developer's different levels of knowledge of the methods. The methods on which a developer has the highest usage and implementation expertise fall into the well known category L1. In addition, the methods on which the developer gained recent expertise may also fall into the L1 category. Category L2 represents the methods which a developer vaguely knows. For example, a developer who has implemented a method knows more about it than the developer who has used it sometimes. Therefore, all the methods implemented and all the methods often used by a developer fall into the L1 category, but the less used methods would fall into the L2 category. Despite that, the implemented methods could also be included in L2 if they were implemented a long time ago by the developer, as knowledge decays with time. L2 also contains the other methods of a class that are neither implemented nor used by a developer, but s/he might possess vague knowledge about them. Category L3 contains the methods that a developer anticipates to exist in the API or project but is not certain about their existence. From the figure, we can see that a portion of L3 lies outside the all methods rectangle L4, which means L3 contains such methods that a developer believes to exist but actually they do not. Besides the developers' knowledge and belief, there exist such methods which are completely unanticipated by them. Those are shown in the [L4 - L3] area of Figure 2.1. Depending on developers' contribution and learning, the methods change their category over time.

2.1.2 Changesets

In the software development life-cycle, a version control or revision control system is a vital component. It keeps track of each and every change made in the source code of a project. The difference between two successive versions or commits in a version control system of a project repository is referred to as a changeset [14, 15]. A changeset contains the change diff of at least one source code file of the project repository. A change diff refers to the chunk of a source code file that is changed through a commit. A changeset can even

consist of the change diffs of all of the files of the repository. Through the change diff, all the changes made in the new version from the previous version can be determined, where the changes could be either insertions or deletions.

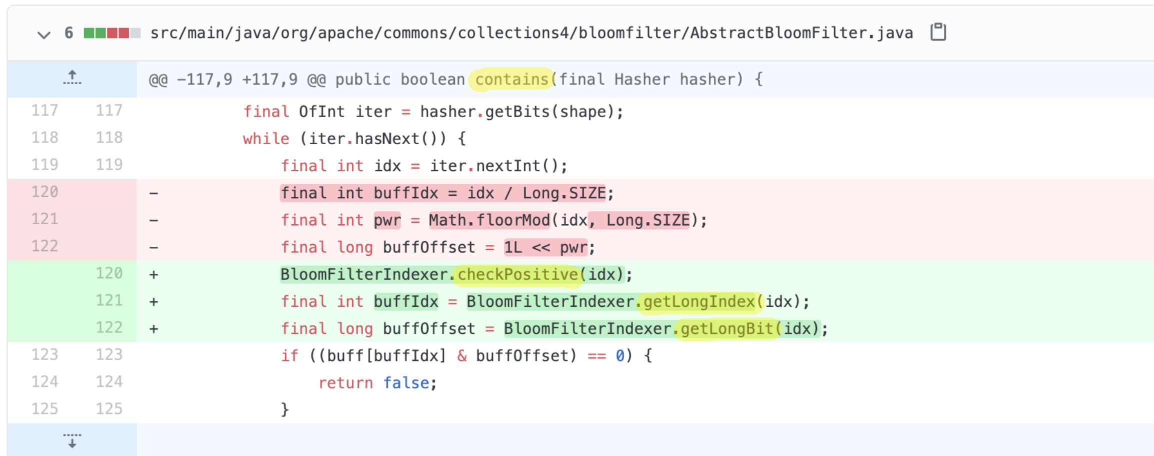


Figure 2.2: A change diff snap from the *Apache Commons Collections* project

Figure 2.2 shows a change diff from the changeset of a commit[2] made to the *Apache Commons Collections* project. All the green lines in the change diff denote insertions, where the red lines denote deletions. The author of this commit *Alex Herbert* has made these changes into the `contains(..)` method of the `AbstractBloomFilter.java` source code file. This implies, he has gained implementation expertise on the method `contains(..)` by making this commit. We can also see three method calls made in the inserted lines. The methods are `checkPositive(..)`, `getLongIndex(..)`, and `getLongBit(..)`. These reflect that *Alex Herbert* has gained usage expertise on these methods through this commit.

2.2 Related Work

Recommending experts is a significant research area in software engineering. This section discusses some previous approaches to recommend experts based on usage expertise, implementation expertise, and general or domain specific expertise.

[2]https://github.com/apache/commons-collections/commit/8b4ecbc084eaf8e61f002612f7ca156fcc1e107e

2.2.1 Usage Expertise Recommenders

Schuler and Zimmerman [10] introduced the concept of usage expertise and proposed a well-formed methodology to determine usage expertise of developers. They were also the first to differentiate between implementation expertise and usage expertise. They created expertise profiles for each developer working on the Eclipse project. The profiles were created by using the changed locations information from the CVS repositories of the project. They claimed that their expert profiles could help to identify experts, support improving collaboration among developers, and analyze API usage of developers. The main difference between their approach and our approach is that they did not resolve the types or signatures of the called methods, which we did. The authors identified method calls only with their names and their number of arguments. The purpose of identifying experts for methods cannot be achieved entirely without knowing their origin, as projects usually have many methods with the same name and even with the same number of arguments. Our recommender system completely resolves the method signatures for most of the in-project methods and all the API methods[3].

Ma et al. [7] extended the previous work of Schuler and Zimmerman [10] by assessing the validity of their concept. They created two types of expertise profiles for each developer: an implementation profile and a usage profile. The first one contained the inserted or modified methods by the developer and their frequency while the latter one contained the inserted method calls by the developer and their frequency. They scored the expertise of the developers using these profiles based on several criteria (such as the frequency, recency, depth, breadth, etc. of changes). They repeated this procedure with every new commit and updated the profiles. To recommend experts for a given query commit (i.e., the commit for which expertise information is requested), they generated expertise profiles for developers with data prior to that commit. Using the profiles and the query commit as input heuristics, they generated an ordered list of expert developers. To evaluate the sys-

[3] See Subsection 4.2.1 for details.

tem, they checked the successful recommendation appearance percentage within the Top-N suggestions made. In addition to exploring their expertise models, we combine usage and implementation expertise to generate a unified expertise model and discover the overall expertise of the developers. The combined metric was mentioned as one of the future works of the authors, but it was never explored. We also examine the impact of the expertise types on the overall expertise of a developer. The authors evaluated their recommender using *AspectJ* and *Eclipse* projects, where *AspectJ* is a plugin or tool for *Eclipse*. Conversely, we evaluated the expertise models with two other more generalized projects, along with *AspectJ*.

2.2.2 Implementation Expertise Recommenders

Anvik and Murphy [1] proposed two approaches for determining implementation expertise for bug reports of a project: one using the source repository check-in logs for the reports and another one using the user information collected from a bug network containing the bug reports. The result from the first approach is the set of all the developers who contributed to the containing module of each changeset file of the bug report. At first, they generated the list of changeset files for a bug report by linking the bug report with a source repository. After that, they determined the containing module, and finally, they generated the list of developers by examining the revision history. On the other hand, the second approach used data from the bug report carbon-copy lists that contained individuals who wanted to be informed about changes in the report, the commenters of that report, and the developer who resolved the report to generate the implementation expertise information. Finally, they filter the results from both the approaches to eliminate irrelevant (i.e., no longer an active project member) developer names.

Both Mockus and Herbsleb [9] and McDonald and Ackerman [8] use a form of the Line 10 Rule to recommend experts for a source code file. The Line 10 Rule refers to a heuristic in which by using the committer name information, the expert developer for a source file

is determined. The committer name is extracted from the 10th line of the commit log for that particular source file. Expertise Browser (ExB) by Mockus and Herbsleb [9] uses Experience Atoms or EAs as the primary unit of expertise. The authors gathered EAs from source code repositories and associated each of them with several socio-technical network domains. Finally, they queried the EAs to find experts and trace their relationships. On the other hand, McDonald and Ackerman [8] use the Line 10 Rule heuristic to create expertise profiles for the organization members in their Expertise Recommender (ER). They query the profiles to recommend experts when a new tech support call comes to the organization.

2.2.3 Domain Specific Expertise Recommenders

XT_{IC} [13] by Teyton et al. supports automatic expert identification by proposing a domain specific language which specifies the skills of developers. It browses software repositories to extract developers' experience level and match them with developers' skills. Lastly, it ranks the developers based on the data collected to identify experts. Teyton et al. showed the efficiency of their system by validating it through stress testing. They also checked the accuracy of their system.

xFinder [6] by Kagdi et al. is a tool that recommends an ordered list of expert developers who have a clear concept and knowledge about any part of a source code on which some other developers need to work. Kagdi et al. generated the ranked list based on several data (such as expertise, experience, and contributions of that developer) from the source repository of the version control system of that particular project. They assumed that the developer who has resolved a similar problem in the past is the most appropriate for doing or helping in the same type of task in the future. The heuristic criteria infers the expertise of the developers, their change activity, and their contribution to the commits of the same context or in a particular file in the past. They have also taken the developer's activity or interactions with the system into consideration to recommend experts.

Steinmacher et al. [12] proposed a recommendation system to recommend a mentor

to new developers who want to start contributing to the source code of an open source project directly or want to solve a bug or resolve an issue. The system used an approach of calculating the socio-technical and current interest scores of the developers to recommend a mentor. They considered the mail lists, the issue tracker, and the available workspace information of the project as inputs while calculating the scores. Finally, the researchers further analyzed the inputs using several analyzers like Relationship Analyzer, Temporal Analyzer, Sociability Analyzer, Workspace Metrics Analyzer, Recent Context Analyzer, and Mentor recommender to make the recommendations.

2.3 Summary

In this chapter, we have discussed the concepts and background information that are relevant to our recommender creation approach. We categorized methods based on developers' knowledge and experience. Additionally, we explained repository changesets and identified developers' expertise information. We also presented the previous research work that is related to our study. Various researchers have followed different ways to determine expert developers for software development projects. Previous researchers explored developers' usage expertise and implementation expertise separately. Although Ma et al. [7] planned to combine the usage expertise and the implementation expertise of developers in the future, they never explored it. Unlike our method-based approach, previous studies determined experts for the files of a project. Furthermore, some researchers explored developer expertise based on developer skills, whereas others suggested general experts who can act as mentors.

Chapter 3

Expertise Recommender Creation Methodology

In this thesis, we explore the expertise of developers for methods of software development projects. This section discusses the methodology used for creating our expert developer recommender system, along with the definitions of developer's method expertise.

3.1 Expertise Definitions

We define the *"expertise"* of a software developer based on their usage and implementation expertise. We define *"expertise"* as follows:

$$E(d,m) = w_1 \times U(d,m) + w_2 \times I(d,m) \quad (3.1)$$

The expertise score E of developer d for a method m is based on the developer's usage expertise *U(d, m)* on the method and implementation expertise *I(d,m)* on the method. The values of w_1 and w_2 are the weights assigned to the developer's usage expertise and implementation expertise respectively, where the weights range from `0` to `1` (see Table 4.2 on Page 35). We assign these weights to distinguish the impact of the individual usage and implementation expertise on the overall expertise of the developers. The values for w_1 and w_2 are set in such a way that if w_1 = `X`, then w_2 = `1 - X`.

For the cases where the implementation contribution of a developer is not found for a method, or w_2 is set to `0`, we use the following form of Equation 3.1:

$$E(d,m) = w_1 \times U(d,m) \tag{3.2}$$

Similarly, for the cases where the usage contribution of a developer is not found for a method, or w_1 is set to `0`, we use the following form of Equation 3.1:

$$E(d,m) = w_2 \times I(d,m) \tag{3.3}$$

3.1.1 Usage Expertise

Usage expertise is the expertise a developer accumulates on a method of a project or a library method by calling or using it within the code they write. The usage expertise of developer *d* for method *m* is defined by the following equation:

$$U(d,m) = \sum_{i=1}^{N} \sum_{j=0}^{M-1} C_{N-i}(us_j(m)) \tag{3.4}$$

where $C_{\mathrm{N-i}}$ represents the $N-i^{th}$ commit from a total number of *N* commits in a project and *M* is the total number of used methods in the $N-i^{th}$ commit by developer *d*. The commits are ordered chronologically from the oldest to the most recent and numbered in the backward sequence (i.e., *0* = most recent, *N-1* = oldest). Therefore, the initial commit of a project is denoted as $C_{\mathrm{N-1}}$ and the latest commit of a project is denoted as C_0. We start working from the initial commit and end at the latest one. The decisive function $us_{\mathrm{j}}(m)$ returns `1` if the jth method call inserted by developer *d* is method *m*, `0` otherwise. We chose to look at insertions only because we believe that insertions are more highly correlated with expertise than deletions. *U (d, m)* is normalized when used for recommendation.

$$us_j(m) = \begin{cases} 1 & for\ used_j = m \\ 0 & for\ used_j \neq m \end{cases}$$

```
16 bcel-builder/src/org/aspectj/apache/bcel/classfile/MethodParameters.java

@@ -29,6 +29,8 @@
29 29        public final static int ACCESS_FLAGS_SYNTHETIC = 0x1000;
30 30        public final static int ACCESS_FLAGS_MANDATED  = 0x8000;
31 31
   32 +      // if 'isInPackedState' then this data needs unpacking
   33 +      private boolean isInPackedState = false;
32 34        private byte[] data;
33 35        private int[] names;
34 36        private int[] accessFlags;
@@ -37,6 +39,7 @@ public MethodParameters(int index, int length, DataInputStream dis, ConstantPool
37 39                super(Constants.ATTR_METHOD_PARAMETERS,index,length,cpool);
38 40                data = new byte[length];
39 41                dis.read(data,0,length);
   42 +              isInPackedState = true;
40 43        }
41 44
42 45        private void ensureInflated() {
@@ -55,17 +58,22 @@ private void ensureInflated() {
55 58                                accessFlags[i] = dis.readUnsignedShort();
56 59                        }
57 60                }
   61 +              isInPackedState = false;
58 62            } catch (IOException ioe) {
59 63                throw new RuntimeException("Unabled to inflate type annotation data, badly formed?");
60 64            }
61 65        }
62 66
63 67        public void dump(DataOutputStream dos) throws IOException {
64 68            super.dump(dos);
65    -          dos.writeByte(names.length);
66    -          for (int i=0;i<names.length;i++) {
67    -              dos.writeShort(names[i]);
68    -              dos.writeShort(accessFlags[i]);
   69 +          if (isInPackedState) {
   70 +              dos.write(data);
   71 +          } else {
   72 +              dos.writeByte(names.length);
   73 +              for (int i=0;i<names.length;i++) {
   74 +                  dos.writeShort(names[i]);
   75 +                  dos.writeShort(accessFlags[i]);
   76 +              }
69 77            }
70 78        }
71 79
```


Figure 3.1: Changeset snapshot from commit #484 of project *AspectJ*

For instance, Figure 3.1 represents a changeset snapshot from commit #484[4] or C_{484} of project *AspectJ*. In this commit, developer *Andy Clement* has used 4 method calls, namely, `write(..)`, `writeByte(..)`, `writeShort(..)`, and again `writeShort(..)`. If we inquire about the usage expertise of developer *Andy Clement* on method `writeShort(..)` at commit C_{484}, the result will be,

$$U(AndyClement, writeShort(..)) = \sum_{j=0}^{3} C_{484}(us_j(writeShort(..))) = (0+0+1+1) = 2$$

Even though the developer has gained usage expertise on all of the called methods at commit C_{484}, the above equation considers all other method calls as 0 except for the inquired one, here `writeShort(..)`. In the same manner, by calculating and adding up the usage expertise score of developer *Andy Clement* on method `writeShort(..)` at each commit of the source code repository, Equation 3.4 will return the total usage expertise score of the developer on that method.

3.1.2 Implementation Expertise

Implementation expertise is the expertise that a developer accumulates on a method of a project by writing the method (i.e., by creating it or modifying it). The implementation expertise of developer *d* for method *m* is defined by the following equation:

$$I(d,m) = \sum_{i=1}^{N} \sum_{j=0}^{M-1} C_{N-i}(imp_j(m)) \tag{3.5}$$

In the equation above, $C_{\mathrm{N-i}}$ represents the $N-i^{th}$ commit from a total number of N commits in a project and M is the total number of implemented methods in the $N-i^{th}$ commit by developer d. The commits are ordered chronologically from the oldest to the most recent and numbered in the backward sequence (i.e., *0* = most recent, *N-1* = oldest).

[4]The commit serial number for commit id `c4f9f951c35f7b7645696ffded594e2dded07476` of project *AspectJ* is 484 as of May 2020, when the total number of commits for the project is 8130.
URL: https://github.com/eclipse/org.aspectj/commit/c4f9f951c35f7b7645696ffded594e2dded07476

Therefore, the initial commit of a project is denoted as C_{N-1} and the latest commit of a project is denoted as C_0. We start working from the initial commit and end at the latest one. The decisive function $imp_j(m)$ returns the `number of insertions` (including comments and blank lines) done, if the jth method implemented by developer d is method m, `0` otherwise. We chose to look at insertions only because we believe that insertions are more highly correlated with expertise than deletions. $I(d, m)$ is normalized when used for recommendation.

$$imp_j(m) = \begin{cases} insertion_count & for\ implemented_j = m \\ 0 & for\ implemented_j \neq m \end{cases}$$

For instance, in Figure 3.1, we can see that developer *Andy Clement* has made the following changes at commit C_{484}:

- 1 insertion (+) and 0 deletions(-) on method `ensureInflated()`
- 8 insertions (+) and 4 deletions(-) on method `dump(..)`

Here, the number of insertions and deletions represents the number of lines inserted into (green lines in the Figure) and deleted from (red lines in the Figure) each method, respectively. Now, if we inquire about the implementation expertise of developer *Andy Clement* on method `dump(..)` at commit C_{484}, the result will be,

$$I(AndyClement, dump(..)) = \sum_{j=0}^{1} C_{484}(imp_j(dump(..))) = (0+8) = 8$$

Even though the developer has gained implementation expertise on both of the implemented methods at commit C_{484}, the above equation considers the actual *insertion_count* for the inquired method `dump(..)` only, and `0` as the *insertion_count* for the other implemented method `ensureInflated()`. In the same manner, by calculating and adding up the implementation expertise score of developer *Andy Clement* on method `dump(..)` at each

commit of the source code repository, Equation 3.5 would return the total implementation expertise score of the developer on that method.

3.2 Recommender Creation Approach

In this thesis, we sought the answer to a research question of whether or not a combined method expertise model outperforms the individual method usage expertise model and method implementation expertise model. This research question led us towards calculating the combined method expertise score, as well as the individual method usage expertise score and method implementation expertise score of a developer. To determine the overall method expertise for a developer, we determined the method usage expertise and method implementation expertise separately. The procedures to determine both of them are similar. We mined both the source code repository and the version history of a software development project to detect the usage and implementation contribution of each developer working on it and then ranked them accordingly. The workflow of our recommender system is illustrated in Figure 3.2.

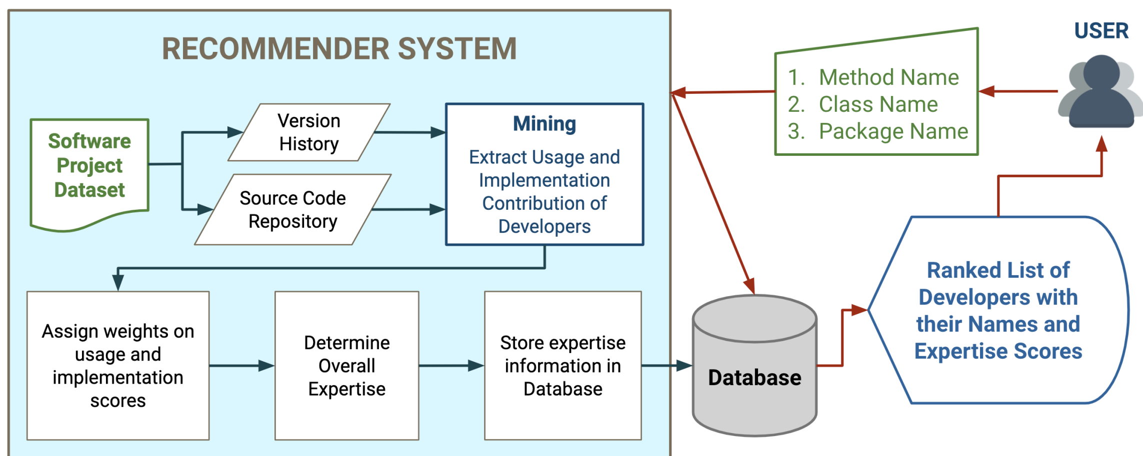


Figure 3.2: The Recommender System Workflow

The following subsections describe the mining approaches to finding both the individual expertise types followed by the score calculation procedure and the combining procedure

to generate the overall developer method expertise.

3.2.1 Mining the Version Control Repository

A version control or revision control system tracks the progress of a software development project over its lifetime. Information recorded includes what changes are made to the codebase, at what time, by which developer, and for what reason. Using the version history data, we extracted the contribution information of each developer working on a project. We examined all the commits, starting from the initial one, up to the one before the *query commit*. The commit, from which a *query method* is selected, is referred to as a *query commit*. Here, *query method* refers to the method for which developer expertise is requested.

From the changesets in the commits of the project's version control system, we identified the method calls made, and the methods added or modified by each developer throughout the project. For each commit, we extracted the changeset, the commit id, the commit message, the details of the developer who made the changes (i.e., the author of the commit), the details of the developer who committed the changes (i.e., the committer of the commit), the authored date, and the committed date. In most cases, the author and the committer of a commit are the same person. However, there are cases where the committer is a project member with commit permissions whereas the author is a developer that made the patch but does not possess commit permissions for that project. This situation mostly occurs in open-source projects where any developer can contribute to a project but only the project members can approve or disapprove the contributions.

We mined the version control system to extract the insertions made into the change diff of each source code file (except the test code files) of the changeset. We detected and saved the insertion line numbers for a particular source code file. As we worked commit-by-commit, we needed to keep synchronization between the version repository and source code repository. We checked out the source code repository to a particular commit version each time before mining, to maintain the synchronization between the two. This ensured

that we are examining the same version of the source code files in the source code repository as the version control repository.

3.2.2 Mining the Source Code Repository

Using the data mined from the version control repository, we mined the source code repository of the dataset project to find developer expertise. For each commit, we mined only those source code files of the source code repository that were changed in that commit. We used a parser library to mine the source code file, which is discussed in detail in subsection 4.2.1. The parser generates an Abstract Syntax Tree (AST) with the information present in the source code files and then resolves the nodes of the AST to identify their root source and relationship between them. We mine the following information about the method declarations (i.e., method implementations) and the method invocations (i.e., method usages) within a file: the method name, class type of its parameters (if any), its return type, the class name in which the method is, and the package name under which the class belongs.

With the help of the resolved data from the parser library, we extracted the usage expertise and implementation expertise information of the developers. To detect usage expertise, at first, we extracted all the method calls made, their root information, and the line numbers they belong to in a source code file. After that, using the insertion line number saved earlier (Subsection 3.2.1), we checked whether the inserted line of that file contains a method call expression or not. If a method call is detected, then we recorded the commit author as a usage expert for that method call. Similarly, to detect implementation expertise, we extracted all the method declarations, their root information, and their line range in a source code file. Then we checked if the inserted lines fall within any method range or not. If found, then we recorded the commit author as an implementation expert for that method.

3.2.3 Calculating Expertise Scores

Once we have all the usage and implementation contribution information of the developers of a project, we calculate their individual usage expertise score, individual implementation expertise score, and overall expertise score for the methods of that project. Before calculating their final scores for a method, their original scores are normalized relative to the maximum score retrieved for any developer on that particular method. The normalization process is discussed below.

Normalizing Expertise Scores

The ranked expert developer list in our recommender system is not generated with the raw expertise scores of the developers. Instead, the usage expertise score and the implementation expertise score for a developer on a method are normalized relative to the maximum usage expertise score and the maximum implementation expertise score achieved by any developer on that method. We normalize the two scores to achieve values in a range from `0` to `10`, meaning that we multiply the values by `10`, after dividing them by their maximum scores. This normalization helps to maintain score consistency throughout the system, and better handle the developer ranking scheme.

Therefore, if *U(m)* is the set of the usage expertise scores of all the developers (*D*) of a project on method *m*,

$$U(m) = \{U(d_0, m),\ U(d_1, m),\ U(d_2, m),\ ....,\ U(d_{D-1}, m)\} \tag{3.6}$$

we normalize the usage expertise of developer *d* on method *m*, *U(d, m)* as:

$$U(d, m) = \frac{U(d, m)}{U(m)_{max}} \times 10 \tag{3.7}$$

Likewise, if *I(m)* is the set of the implementation expertise scores of all the developers (*D*) of a project on method *m*,

$$I(m) = \{I(d_0,m),\ I(d_1,m),\ I(d_2,m),\ ....,\ I(d_{D-1},m)\} \tag{3.8}$$

we normalize the implementation expertise of developer *d* on method *m*, *I(d, m)* as:

$$I(d,m) = \frac{I(d,m)}{I(m)_{max}} \times 10 \tag{3.9}$$

For example, if {`12, 4, 0, 19, 5, 2, 24, 1`} is the set of original usage or implementation expertise scores of all the developers of a project on method *m*, then the normalized usage or implementation expertise score set will be {`5, 1.67, 0, 7.92, 2.08, 0.83, 10, 0.42`}, where all the elements are in the range from `0` to `10`.

3.2.4 Determining Overall Expertise

To calculate the overall expertise score of a developer for a method, we use the normalized usage and implementation expertise scores of that developer for that method. We compute the overall expertise of developer *d* on method *m*, *E(d, m)*, by using Equation 3.1, where the weights w_1 and w_2 range from `0` to `1`. The values for w_1 and w_2 are set in such a way that if w_1 = `X`, then w_2 = `1 - X`.

In some cases, the usage contribution of a developer is not found for a method, but only the implementation contribution is found. This happens when a developer possesses no history of using or calling that method but has a history of implementing (i.e., creating or modifying) it. For such methods, only the implementation expertise score of the contributing developer is found, and therefore, we used Equation 3.3 to calculate the overall expertise score. In the same way, the implementation contribution of a developer may not exist for a method. Therefore, only the usage instances contribute to the developer's overall expertise for that method. This could happen in two situations. The first is for API or library methods, where an in-project developer's implementation contribution cannot be determined, as the implementation is outside of the project's scope. Second, for some project methods, a developer may possess no history of implementation (i.e., creation or modification), but

only that of its usage. For such methods, only the usage expertise score of the contributing developer is found, and therefore, we used Equation 3.2 to calculate the overall expertise score (see Appendix A).

3.3 Recommending Experts

Having created the recommendation system, we can provide users with a ranked list of expert developers for a method. For our recommendation system, the user inputs the project name, the method name from that project for which s/he wants to get the expert developer list, the class name where the method is in, and finally, the package name in which the class is. The recommender system returns three ranked lists of developers with their expertise score. The three lists are the usage expertise list, implementation expertise list, and overall expertise list.

```
Please enter your project name: commons-collections
Please enter the method name from project 'commons-collections': hasNext
Please enter the class name of method 'hasNext': ArrayIterator
Please enter the package name of class 'ArrayIterator': org.apache.commons.collections

~~~~Successfully retrieved expert developers.~~~~

Developer Name ::: Usage Expertise Score
-----------------------------------------

1. Morgan James Delagrange ::: 10.0
2. James Strachan ::: 10.0
======================================

Developer Name ::: Implementation Expertise Score
--------------------------------------------------

1. James Strachan ::: 10.0
2. Morgan James Delagrange ::: 2.5
3. Michael Smith ::: 0.83
======================================

Developer Name ::: Overall Expertise Score
-------------------------------------------

1. James Strachan ::: 10.0
2. Morgan James Delagrange ::: 6.25
3. Michael Smith ::: 0.415
======================================
```

Figure 3.3: Recommendations for method `hasNext` from project `commons-collections`

Figure 3.3 shows an example of an implementation of the approach in action where the recommender provides results for method `hasNext` of class `ArrayIterator` under package `org.apache.commons.collections` from the project `apache/commons-collections`. After collecting the necessary information from the user, the recommender provides the developer with expertise information for that method. In this case, for the method `hasNext`, the recommender provides two developers for Usage Experts, three developers for Implementation Experts, and three developers as Overall Experts. The set 'overall expert developers' is the union of the set 'usage expert developers' and the set 'implementation expert developers'. As we can see, developers *James Strachan* and *Morgan James Delagrange* have acquired both usage and implementation expertise for method `hasNext`, where developer *Michael Smith* has acquired only implementation expertise on the method. The values of w_1 and w_2 are set to `0.5` in this example while calculating the overall expertise.

In the case of method overloading, the recommender provides the users with a list of all the methods found and asks the user to choose the intended one from the list. See Appendix A for this and further examples.

3.4 Summary

In this chapter, we have presented how we determine usage and implementation expertise for developers of a particular software project. To create the expertise recommender, we mined the version control repository and the source code repository of a project to extract the developers' contribution to the project over its lifetime. After finding the usage instances and implementation contributions of the developers for the methods of the project, we combined the values for the two types of expertise to determine the developers' overall method expertise.

Chapter 4

Evaluation Methodology

To evaluate the results of an expert developer recommendation system (i.e., to calculate precision and/or recall), the actual expert developers must be known. The expertise of the developers can be analyzed by their knowledge and experience with the system. There can be several ways to observe the knowledge and experience of the developers. The most accurate way would be asking or obtaining feedback from the developers directly. However, this is not feasible in most cases. Examining the usage and implementation expertise could be considered a feasible way to determine expert developers, which we have done in our approach.

Apart from working on the project, developers can accumulate knowledge and experience by reading the project documentation, getting suggestions from or questions answered by other experienced developers, learning from forums, lectures, and tutorials, and even studying the existing codebase or other relevant code. Most of these observation sources require access to the browsing history of the developers, which is also not easily obtainable. Therefore, we validated our recommender system using the evaluation strategy used by Ma et al. [7] and compared our results with theirs. We conducted an analytical evaluation of our recommender system to seek the answers to the following research questions:

RQ1: Does our combined expertise model outperform our individual expertise models?

RQ2: Does the project and development team size affect the overall results?

RQ3: What combination of usage and implementation expertise produces better results?

RQ4: Do our individual and combined expertise models outperform the previous approach by Ma et al. [7]?

The rest of this chapter describes the software project dataset we used for evaluation, the evaluation strategy we adopted, and the computational metrics we used to obtain the evaluation results.

4.1 Data Collection

We used three open-source software development projects of different sizes and domains from *GitHub*[5] to evaluate our expertise models. As our approach requires access to both the version history and the source code repository of a software project dataset, we found open-source projects as the most rational choice. More specifically, we chose to use projects that use the `Java` programming language. Our chosen projects are *Commons Collections*[6] from *Apache*, *Guava*[7] from *Google*, and *AspectJ*[8] from *Eclipse*. Additionally, as we experimented with the effect of project and development team size in the overall recommendations in this thesis (i.e., Research Question #2), our dataset projects vary in the number of contributors and number of commits.

Apache Commons Collections is an extension of the *Java Collection Framework*, which handles collection types in `Java` [2]. *Guava* is a popular library of *Google*, which is comprised of a variety of Google's `Java` core libraries such as `map, set, primitives, graph,` etc. [5]. *AspectJ* is an extension plug-in created for *Eclipse* to work with Aspect-Oriented Programming in `Java` [3]. That the projects are from three different organizations, increases the likelihood that the coding conventions are not similar. As a consequence, it demonstrates the generalizability of our recommender creation approach.

Table 4.1 lists the details of our dataset projects used for evaluating our approach. Based

[5] GitHub is a commonly used online web service for projects that use the Git version control system.
[6] https://github.com/apache/commons-collections
[7] https://github.com/google/guava
[8] https://github.com/eclipse/org.aspectj

on the number of contributors, we categorized *Google's Guava* as a large project, *Apache Commons Collections* as a medium project, and *Eclipse's AspectJ* as a small project. The ratio of number of contributors and number of commits also justifies this categorization.

Table 4.1: Details of the Software Project Datasets (as of July 2020)

Project Name	Java Code	Contributors	Commits	Methods	Contributors : Commits
guava	100%	239	5,286	30,926	1 : 22
collections	99.3%	50	3,388	20,254	1 : 68
aspectj	77.2%	6	8,130	53,643	1 : 1355

Before training the recommender, we filtered out all non-Java (i.e., do not have the extension .java) files from the dataset projects. We also disregarded the .java test code files as they are not part of the main project. This resulted in a dataset with more than one hundred thousand (104,823) methods, where more than fifty thousand of them belonged to *AspectJ*, more than thirty thousand belonged to *Guava*, and the rest of them belonged to *Commons Collections*.

4.2 Evaluation Procedure

To test our recommender to generate the ranked expert developers list for any method from a particular commit, we trained our system with the data up to the commit preceding the query commit. We considered a recommendation successful when the actual commit author of that method's usage or implementation is listed within the Top-N recommendation results [7]. We determined the accuracy of our system by calculating the percentage of successful recommendations within the Top-N results for the methods of a commit. In addition to that, we calculated the Mean Reciprocal Rank (MRR) for the recommendations

made for the methods of a commit by using the ranks of the recommended developers. Both of these evaluation metrics are discussed in detail later in this chapter.

Ma et al. [7] evaluated their recommender using only two projects: *AspectJ* and *Eclipse*, where *AspectJ* is a plugin for *Eclipse*. To compare our results with the original authors' results, we also evaluated our system with the *AspectJ* project from *Eclipse*. Furthermore, we used two other projects, which are *Commons Collections* from *Apache* and *Guava* from *Google* (see Section 4.1). The complete evaluation procedure for the methods of a commit is comprised of the following steps:

1. Training the recommender with expertise data for commits prior to the query commit
2. Gathering expertise data for the query methods
3. Generating recommendations for the query methods
4. Comparing recommendations with original commit author
5. Investigating unsuccessful recommendations and removing false negatives
6. Computing metrics (accuracy and MRR) for the query commit

4.2.1 Training the Recommender

As mentioned earlier, when our recommender system is asked to recommend expert developers for a called or implemented method from a particular commit of a project, we train the system with the contribution information of the developers of that project, that was made before the query commit. Recall that in Figure 3.1 (see Page 16), developer *Andy Clement* implemented two methods and used four method calls in commit #484 of project *AspectJ*. To recommend expert developers for any of these six methods, we train our system with all the repository changeset information from the initial commit (commit #8129[9] as per our dataset, commit date: December 16, 2002) to commit #485[10] (commit date: June

[9] https://github.com/eclipse/org.aspectj/commit/3cde920c3f7eb8241bf569007e25225d80b43c0f
[10] https://github.com/eclipse/org.aspectj/commit/4750cace2d5d2adc817ffe1ff192027b9def283a

12, 2014). By doing this, we use all the expertise information from the past[11]. Depending on the project size and number of commits, the training period runs from *10 to 18 hours*.

Extracting Version Control Data

As our dataset projects are from *GitHub*, we used the `Python` library `GitPython`[12] to extract the `Git` repository data of the projects. First, we derive the changeset from each commit of a project. After that, we extract the insertions made into the change diff of each source code file (except the test code files) of the changeset, along with the inserted line numbers, using `Python`'s string operations.

Parsing Source Code Data

As we evaluate our approach using `Java` projects, we used a `Java` parser library to parse the source code files. We used the library named `JavaSymbolSolver` [11], in the parser library `JavaParser`[13]. `JavaParser` generates an Abstract Syntax Tree (AST) with the information present in the `Java` source code files. `JavaSymbolSolver` resolves the nodes of the generated AST to identify their root source and relationship between them. In simple words, `JavaSymbolSolver` retrieves detailed information about every symbol or element present in a source code file.

Parsing method declarations: First, we work with the method declarations (i.e., method implementations) within each changed file in a commit. Using `JavaParser`, we list all of the methods within a file, then use `JavaSymbolSolver`'s `resolveDeclaration` to resolve the listed methods to get the desired properties for the methods (see Subsection 3.2.2 for details). Despite providing accurate information for most of the method declarations, there were some cases when the symbol solver could not resolve the methods' class names and/or package names. For such method declarations, we split the file path of the methods' class

[11] We examined with both time-windows (from the latest six months to one year) and range-windows (from the latest 100 to 1000) for commits and found that the amount of data was too few to make reasonable recommendations.

[12] https://gitpython.readthedocs.io/en/stable/intro.html

[13] https://github.com/javaparser/javaparser

to derive class names and package names.

Parsing method invocations: After collecting information about all of the method declarations, we work with the method invocations (i.e., method usages) within the changed file. Utilizing the same `resolveDeclaration` functionality of `JavaSymbolSolver`, we resolve the method invocations to get the desired properties. However, there were cases in our experimentation where a resolution could not be made. The cases were more for method invocations than for method declarations. After manual inspections of around ten source code files from each project, we determined that most of these situations arise when there are invocations to static methods, nested methods, nested class methods, some in-class methods, or if the method invocation is nested itself. Figure 4.1 presents a method named `showMessageHandlerPanel(..)` from the source code file *MessageHandlerPanel.java*[14] of project *AspectJ*, which shows the occurrences of both a nested method declaration and several nested method invocations.

To address the above mentioned situations, we do the following:

- For exceptions caused by invocations to the nested class methods under the same calling class, we examine all of the class declarations within the AST members for the changed source code file. We fetch the method declarations for the class declarations to see if any of them matches the method name for which the exception occurred. If the name matches, then we resolve the method and save the method's root and property information.

- For exceptions that occurred as the result of invocations to the static or non-static methods under the same calling class, we examine all of the method declarations within the AST members to see if the invoked method is one of the in-class static or non-static methods. If the method names match, then we resolve the method and save its root and property information.

[14]https://github.com/eclipse/org.aspectj/blob/master/ajbrowser/src/main/java/org/aspectj/tools/ajbrowser/ui/swing/MessageHandlerPanel.java

- Finally, when exceptions are not resolved using the above strategies, we examine the field or variable declarations within the changed file. We fetch all of the variable declarations and match the variable names with that of the method calling variable. If the variable names match, then we resolve that variable and save the class name and package name information for the called method. However, the return type and parameter information cannot be extracted for such cases.

```
39
40         public void showMessageHandlerPanel(BrowserMessageHandler handler, boolean showPanel) {
41                 if (!showPanel) {
42                         setVisible(false);
43                         return;
44                 }
45                 createList(handler.getMessages());
46
47                 try {
48                         jbInit();
49                 } catch (Exception e) {
50                         e.printStackTrace();
51                 }
52                 list.setModel(listModel);
53
54                 MouseListener mouseListener = new MouseAdapter() {
55                         public void mouseClicked(MouseEvent e) {
56                                 if (e.getClickCount() >= 1) {
57                                         int index = list.locationToIndex(e.getPoint());
58                                         if (listModel.getSize() >= index && index != -1) {
59                                                 IMessage message = (IMessage) listModel
60                                                                 .getElementAt(index);
61                                                 Ajde.getDefault().getEditorAdapter().showSourceLine(
62                                                                 message.getSourceLocation(), true);
63                                         }
64                                 }
65                         }
66                 };
67                 list.addMouseListener(mouseListener);
68                 list.setCellRenderer(new CompilerMessagesCellRenderer());
69                 setVisible(showPanel);
70         }
71
```

Nested Method Invocation (line 45)
Nested Method Declaration (lines 55–65)
Nested Method Invocation (line 57)
Nested Method Invocation (lines 61–62)

Figure 4.1: Method with nested method declaration and invocations

We successfully resolved more than `94%` of the dataset methods using these tactics, along with employing the parser library `JavaSymbolSolver`. However, there existed a handful of methods that we could still not resolve. To be precise, around `6%` (5,772) of the

total number of methods (104,823) in our dataset remained unresolved[15]. We excluded the unresolved methods at the time of evaluating our system to avoid false negatives.

4.2.2 Gathering Expertise Data

Once we have the trained data for the past commits of the query commit, we gather the expertise data from them. We collect expertise data only for the methods of that query commit. Simply put, for each method of the query commit, we collect the usage and implementation contribution of the developers who have worked on those methods before the query commit. For the same example of Figure 3.1 (see Page 16), to prepare recommendations for any of the six changed methods from commit #484 of *AspectJ*, we collect the contribution information of the developers on the query method, from commit #8129 to commit #485.

4.2.3 Generating Recommendations

After collecting the necessary expertise data, we generate the recommendations for a query method. We generate both the usage expertise recommendation list and the implementation expertise recommendation list, and then we produce the combined expertise recommendation list for the query method.

Usage and Implementation Expertise Recommendations

Both the usage expertise recommendation list and the implementation expertise recommendation list have the same format: the name of the recommended developer and their normalized expertise score. The recommendations are presented in descending order (i.e., the developer with the highest score at the top of the list). The usage and implementation expertise scores are calculated using Equation 3.4 and 3.5, respectively.

Figure 4.2 gives an example of the usage expertise recommendations and the implementation expertise recommendations made for the method named `keySet` from commit

[15] What percentage of unresolved methods would result in an ineffective recommender was unexplored.

```
Evaluating commit no. 3294 (9e513787c859c6ea922574ef30c66f17eb9158f1) ---- with 22 methods
Author: Chris Povirk, Committer: Chris Povirk

~~~~~~~~~~~ Working with method: keySet ~~~~~~~~~~~

Developer Name ::: Usage Expertise Score
-----------------------------------------

1. Kurt Kluever ::: 10.00 (9)
1. Charles Fry ::: 10.00 (9)
2. kevinb@google.com ::: 8.89 (8)
3. Chris Povirk ::: 7.78 (7)
4. Christian Gruber ::: 6.67 (6)
5. Colin Decker ::: 5.56 (5)
6. guava.mirrorbot@gmail.com ::: 3.33 (3)
6. Louis Wasserman ::: 3.33 (3)
7. Christian Gruber ::: 2.22 (2)
=========================================

Developer Name ::: Implementation Expertise Score
--------------------------------------------------

1. Kurt Kluever ::: 10.00 (9)
2. kevinb@google.com ::: 7.78 (7)
3. Charles Fry ::: 4.44 (4)
4. Christian Gruber ::: 3.33 (3)
4. Louis Wasserman ::: 3.33 (3)
5. Chris Povirk ::: 2.22 (2)
6. guava.mirrorbot@gmail.com ::: 1.11 (1)
=========================================
```

Figure 4.2: Usage and Implementation Expertise Recommendations for method `keySet` from commit #3294 of project `guava`

#3294[16] of project `guava`. The method `keySet` is one of the 22 methods that are changed (either used or implemented) in commit #3294. The author and committer of this commit is *Chris Povirk*. The recommendations have been produced using the data from commit #5285[17] (i.e., the initial commit, commit date: June 18, 2009) to commit #3295[18] (commit date: July 29, 2013). In other words, the suggested expert developers have worked on `keySet` during the period. The number inside the parenthesis beside an expertise score is the raw expertise score prior to normalization. Note that, the developers with equal expertise scores are ranked equally (e.g., Kurt and Charles for Usage, Christian and Louise for Implementation in Figure 4.2).

[16] https://github.com/google/guava/commit/9e513787c859c6ea922574ef30c66f17eb9158f1
[17] https://github.com/google/guava/commit/db3929bd79ef57f8da40515b663f40200a941fa8
[18] https://github.com/google/guava/commit/8ea0f20aac47a603c01c1fe895777df0f3afbedf

Overall Expertise Recommendations

The usage and implementation expertise recommendation results are used to produce the overall expertise recommendations for the query method. The overall expertise score of a developer for a method is calculated using Equation 3.1. To determine the best values for w_1 and w_2, we investigated five different weights for the usage and implementation expertise scores. The weight values were chosen to understand the impact of the individual expertise scores on the overall expertise of the developer for that method. The weights range from 0 to 1 as shown in Table 4.2. By changing the weights from 0 to 1, we examine whether both of the expertise types are equally important for a developer's overall expertise or if any one of them has more significance. Note that when $w_1 = 0$ and $w_2 = 1$ (i.e., `WeightSet-1`), the overall expertise score is the implementation expertise and vice versa for $w_1 = 1$ and $w_2 = 0$ (i.e., `WeightSet-5`).

Table 4.2: Weights assigned on the Usage and Implementation expertise scores

Sl No.	Usage Weight	Implementation Weight
1	0	1
2	0.25	0.75
3	0.50	0.50
4	0.75	0.25
5	1	0

Figure 4.3 shows the overall expertise recommendations for the method `keySet` from commit #3294 of project `guava`. The five ranked lists show the results for the five combinations of weights (shown at top of each list). Note that the list in Figure 4.3a is the same as the implementation expertise list in Figure 4.2 and similarly, the list in Figure 4.3e is the same as the usage expertise list in Figure 4.2. Figure 4.3b, 4.3c, and 4.3d show the results for the other three weight sets.

```
Evaluating commit no. 3294 (9e513787c859c6ea922574ef30c66f17eb9158f1) ---- with 22 methods
Author: Chris Povirk, Committer: Chris Povirk

~~~~~~~~~~~ Working with method: keySet ~~~~~~~~~~~

Developer Name ::: Overall Expertise Score
-------------------------------------------
w1 = 0.0 and w2 = 1.0

1. Kurt Kluever ::: 10.00
2. kevinb@google.com ::: 7.78
3. Charles Fry ::: 4.44
4. Louis Wasserman ::: 3.33
4. Christian Gruber ::: 3.33
5. Chris Povirk ::: 2.22
6. guava.mirrorbot@gmail.com ::: 1.11
======================================
```

(a) For `WeightSet-1`

```
w1 = 0.25 and w2 = 0.75

1. Kurt Kluever ::: 10.00
2. kevinb@google.com ::: 8.06
3. Charles Fry ::: 5.83
4. Chris Povirk ::: 3.61
5. Louis Wasserman ::: 3.33
6. Christian Gruber ::: 3.06
7. Christian Gruber ::: 1.67
7. guava.mirrorbot@gmail.com ::: 1.67
8. Colin Decker ::: 1.39
=======================================
```

(b) For `WeightSet-2`

```
w1 = 0.5 and w2 = 0.5

1. Kurt Kluever ::: 10.00
2. kevinb@google.com ::: 8.33
3. Charles Fry ::: 7.22
4. Chris Povirk ::: 5.00
5. Christian Gruber ::: 3.33
5. Louis Wasserman ::: 3.33
6. Colin Decker ::: 2.78
6. Christian Gruber ::: 2.78
7. guava.mirrorbot@gmail.com ::: 2.22
=======================================
```

(c) For `WeightSet-3`

```
w1 = 0.75 and w2 = 0.25

1. Kurt Kluever ::: 10.00
2. Charles Fry ::: 8.61
2. kevinb@google.com ::: 8.61
3. Chris Povirk ::: 6.39
4. Christian Gruber ::: 5.00
5. Colin Decker ::: 4.17
6. Louis Wasserman ::: 3.33
7. guava.mirrorbot@gmail.com ::: 2.78
8. Christian Gruber ::: 2.50
=======================================
```

(d) For `WeightSet-4`

```
w1 = 1.0 and w2 = 0.0

1. Kurt Kluever ::: 10.00
1. Charles Fry ::: 10.00
2. kevinb@google.com ::: 8.89
3. Chris Povirk ::: 7.78
4. Christian Gruber ::: 6.67
5. Colin Decker ::: 5.56
6. guava.mirrorbot@gmail.com ::: 3.33
6. Louis Wasserman ::: 3.33
7. Christian Gruber ::: 2.22
=======================================
```

(e) For `WeightSet-5`

Figure 4.3: Overall Expertise Recommendations for method `keySet` from commit #3294 of project `guava`

As may be expected, some items change rank or may not appear depending on the weighting. For example, *Colin Decker* does not appear in Figure 4.3a as the developer only has usage expertise for `keySet`. Similarly, *Chris Povirk* changes ranks in Figure 4.3a, 4.3b, and 4.3d as he has both usage and implementation expertise and the different weightings affect their overall expertise score.

4.2.4 Comparing Recommendations

Recall that a recommendation result is only considered successful if the commit author of the query method is listed within the Top-N recommendations. However, for open-source projects, it can occur that the commit author is not present in the Top-N recommendations as open-source projects often have a large number of intermittent contributors (i.e., contributors that only make a small number of contributions). In other words, there exists a set of developers who do not work consistently throughout the project or just work on a particular feature or issue. Thus, they do not generate enough development history to be recommended as an expert developer for a module. To address this problem, when evaluating a recommendation list, we looked for the committer of the query commit in the Top-N recommendations when a commit author is not found in the recommendations. The rationale is that the committer possesses at least an equal amount of expertise as the commit author, as common software development practice has that a committer must review the code of the author before merging it into the existing software. To distinguish between these two cases, we searched the recommendation lists using the developers' names and email addresses and stopped searching as soon as a match was found. We labelled the successful recommendations as 'successful-author' or 'successful-committer' depending on the matched developers' roles. The recommendations for which no match was found within Top-N, either with commit author details or the committer details, we considered as 'unsuccessful'.

When evaluating our recommender system, we chose to use three different values for

N: Top-1, Top-5, and Top-10. The recommendations presented in Figure 4.2 (see Page 34) show that the commit author *Chris Povirk* is recommended in rank 3 in the usage expertise list and rank 5 in the implementation expertise list. This implies that both the recommendations are successful for Top-5 and Top-10 benchmarks but unsuccessful for Top-1. Similarly, the overall expertise recommendations with the five different set of weight values presented in Figure 4.3 (see Page 36) show that the commit author *Chris Povirk* successfully appears within the Top-5 and Top-10 recommendations made for method `keySet`, yet is unsuccessful for Top-1. The position of the developer in the ranked lists changes according to the weights assigned. The ranked lists clearly show that the higher weight assigned to the usage expertise score of the developer (i.e., w_1), the better the position the developer secures in the lists due to having a higher usage expertise score (see Figure 4.2 on Page 34).

4.2.5 Investigating Unsuccessful Recommendations

The unsuccessful recommendations do not always represent a failure of the recommender. Depending on the developers' contribution history and the project's timeline, there can be *false negative results*. Here, by *false negative results*, we mean the recommendation results that are labelled as 'unsuccessful' but are not actually unsuccessful. False negatives are addressed by using filtering.

When a recommendation made for a query method is labelled as unsuccessful, we apply a 4-step filtration process on the method to verify the credibility of the recommendation.

i. Recall that we could not resolve around `6%` of our dataset methods due to the parser library failure (Subsection 4.2.1). We examine whether or not the query method is one of those methods that were unresolved by the system. If so, then we do not consider the method from the query method list and ignore the recommendation evaluation results for that method, as it is not appropriate to assess the accuracy for a method for which the system does not have information.

ii. We filter out query methods for which no recommendations are made. This can

occur if and only if the query method is introduced to the system through the query commit. In other words, the method is new and has no history with which to make a recommendation.

iii. For implementation expertise, if the query method is an API or a library method, we disregard the implementation expertise recommendation evaluation result for that method. We examine the package name for a method's class to determine if the method is an in-project method or a library method. For example, if a method's package name has the prefix '`java.`', we label it as a `Java` library method and remove from the implementation expertise query method set. We remove such methods as the implementation information for these methods is not available in the dataset.

iv. To address the incorrect recommendations caused by new developers or less experienced developers, we filter out the methods, for which the recommender does not have sufficient information about the query commit author. We determine sufficient information by calculating the average number of commits each developer should ideally make at any commit level. If the commit author's actual commit count is less than this average commit count threshold, we disregard any unsuccessful recommendations made for that developer at that commit level and exclude those query methods. Conversely, if the developer's history shows an equal or greater number of commits than the average commit count threshold, we deem this an unsuccessful recommendation result.

4.2.6 Computing Evaluation Metrics

We calculate two different evaluation metrics, Accuracy [16] and Mean Reciprocal Rank [17] to measure the performance of our recommender system.

Accuracy

To determine the accuracy of our recommender system, for each query commit, we use the following equation [16]:

$$Accuracy = \frac{SR_{count}}{QM} \times 100\% \tag{4.1}$$

where, SR_{count} is the number of query methods with successful recommendations (SR) among the total number of query methods (QM) for a given query commit.

Then, we calculate the average accuracy for the query methods of a project according to Equation 4.2.

$$Method_{acc} = \frac{1}{QC} \sum_{i=0}^{QC-1} Accuracy_i \tag{4.2}$$

where, $Accuracy_{\text{i}}$ denotes the accuracy for each query commit of the project and QC denotes the total number of query commits for the project.

In addition to average method-wise accuracy, we compute the percentage of query commits in the project with successful recommendations according to Equation 4.3.

$$Commit_{acc} = \frac{number\ of\ commits\ with\ SR}{QC} \times 100\% \tag{4.3}$$

These calculations are done for each of Top-1, Top-5, and Top-10 recommendations.

Mean Reciprocal Rank

The Mean Reciprocal Rank (MRR) [17] for each query commit is the average of the reciprocal value of the rank for each correctly recommended developer for the query methods. We denote the commit-based MRR as MRR_{commit} and determine it according to following equations:

$$MRR_{commit} = \frac{1}{QM} \sum_{i=0}^{QM-1} \frac{1}{rank_i} \tag{4.4}$$

We also compute MRR_{method}, which uses the following equation:

$$MRR_{method} = \frac{1}{QC} \sum_{i=0}^{QC-1} \frac{1}{rank_i} \quad (4.5)$$

For both Equations 4.4 and 4.5, $rank_i > 0$ and $reciprocal_rank_i = \frac{1}{rank_i}$ for successful recommendations. On the other hand, $reciprocal_rank_i$ or $\frac{1}{rank_i} = 0$ for incorrect recommendations.

4.3 Summary

In this chapter, we have discussed the evaluation procedure we used to validate our recommender system and the software projects we used as dataset for the validation. When the commit author or the committer of a method's usage or implementation appeared within the Top-N recommendation for that method, we considered the recommendation successful. Using three dataset projects of three different sizes from three different organizations, we evaluated our recommender system. We computed accuracy and Mean Reciprocal Rank of the recommender for these projects. The results of these computations are discussed in the next chapter.

Chapter 5

Results and Discussion

This chapter presents the results of the evaluation of our recommender system. It also discusses the findings from our experiments to answer the research questions of this thesis. Finally, this chapter ends with a discussion about the threats to the validity of our work.

5.1 Results

We have plotted the results of our commit-by-commit evaluation in charts for each expertise model. The graphs, the average accuracy data, and the MRR experimentation details are presented below for each of the dataset projects.

We have used the following abbreviations and short names in the result presentations:

- **UE** denotes *Usage Expertise* Model Results
- **IE** denotes *Implementation Expertise* Model Results
- **CE-2** to **CE-4** denote *Combined Expertise* Model Results for `WeightSet-2` to `WeightSet-4`, respectively (see Table 4.2 on Page 35)
- **`guava`** denotes dataset project *Google's Guava*
- **`collections`** denotes dataset project *Apache Commons Collections*
- **`aspectj`** denotes dataset project *Eclipse's AspectJ*

5.1.1 Accuracy

We calculated the accuracy of our recommender system for each query commit of the dataset projects using Equation 4.1. With the accuracy results for each commit, we determined the average method-wise accuracy, $Method_{acc}$ using Equation 4.2 and average commit-wise accuracy, $Commit_{acc}$ using Equation 4.3. The average method-wise and commit-wise accuracy results for the three different dataset projects for Top-1, Top-5, and Top-10 recommendations, for both of the individual expertise models and the combined expertise model, are presented below. The accuracies for the combined expertise model with `WeightSet-1` and `WeightSet-5` are deliberately excluded from the results below, as they are the equivalent to those of the implementation expertise model and the usage expertise model, respectively (see Table 4.2 on Page 35).

Average Method-wise Accuracy Results

The average method-wise accuracy results reflect the mean percentage of query methods in each project, for which successful recommendations are made. In other words, it shows the probability of getting successful recommendations for query methods in a project. Tables 5.1, 5.2, and 5.3 show the $Method_{acc}$ results for the dataset projects for Top-1, Top-5, and Top-10 recommendations, respectively.

Table 5.1: Average Method-wise Accuracy Results (Top-1)

Project Name	**UE**	**IE**	**CE-2**	**CE-3**	**CE-4**
`guava`	40.38%	37.68%	40.00%	40.30%	40.76%
`collections`	53.41%	56.46%	53.52%	53.53%	53.56%
`aspectj`	65.41%	65.94%	62.54%	62.66%	64.69%

Table 5.2: Average Method-wise Accuracy Results (Top-5)

Project Name	UE	IE	CE-2	CE-3	CE-4
guava	73.50%	56.42%	71.56%	71.65%	71.69%
collections	76.45%	64.95%	72.67%	72.67%	72.67%
aspectj	87.87%	76.88%	90.10%	90.10%	90.10%

Table 5.3: Average Method-wise Accuracy Results (Top-10)

Project Name	UE	IE	CE-2	CE-3	CE-4
guava	77.82%	56.53%	74.60%	74.62%	74.63%
collections	79.16%	64.95%	74.44%	74.44%	74.44%
aspectj	87.95%	76.88%	90.18%	90.18%	90.18%

Analysis of Results

From the results shown in the tables above, we can see that for the *Guava* project, the usage expertise model and the combined expertise model generate a 40% average accuracy, when we consider Top-1 recommendation only (Table 5.1). The implementation expertise model, nonetheless, generates an average accuracy close to the others (37%). This trend is followed in both of the Top-5 and Top-10 recommendations, except for the UE model which produces slightly higher accuracy than the CE model, and the IE model showing less improvement compared to the others (Tables 5.2 and 5.3).

Similarly, for the *Commons Collections* project, the UE model produces slightly better accuracy than the CE model for Top-5 and Top-10 recommendations, where the IE model trails behind the others. Contrarily, the IE model provides the highest accuracy (56.46%) for Top-1 recommendation for this project, where the others produce slightly less accuracy (53%) than this model.

Unlike the other two projects, where the UE model produced the highest accuracies most of the time, the *AspectJ* results follow a different pattern. For *AspectJ*, the CE model produces the best results (more than 90%) for Top-5 and Top-10 recommendations and very close to the best with WeightSet-4 (64.69% at CE-4) for Top-1 recommendation. Interestingly, for this project also, the IE model generates the highest accuracy (65.94%) for Top-1 recommendation like the *Commons Collections* project. However, for Top-5 and Top-10 recommendations, the UE model once again outperforms the IE model like with *Guava* and *Commons Collections* and produces better accuracy.

Table 5.4: Average Method-wise Accuracy Results Across All Projects

Expertise Type	Top-1	Top-5	Top-10
UE	53.07%	79.27%	81.64%
IE	53.36%	66.08%	66.12%
CE-2 (25% UE, 75% IE)	52.02%	78.11%	79.74%
CE-3 (50% UE, 50% IE)	52.16%	78.14%	79.75%
CE-4 (75% UE, 25% IE)	53.00%	78.15%	79.75%

If we look at accuracy results for these models, generalized across all of the projects (see Table 5.4), we see that we achieved accurate recommendations for more than 50% of the query methods on average when we consider accuracy for Top-1. The percentage increased to 66%-79% when we consider accuracy for Top-5, and to almost the same average percentage (66%-81%) when we look at accuracy for Top-10. The results also show that the UE model outperforms the IE and the CE models when Top-5 and Top-10 expert developers are recommended. This is because we found more method usage instances per developer than method implementation instances when we collected expertise data for Top-5 and Top-10 recommendations. On the other hand, the IE model is the best option to look at when the most expert developer for a query method (Top-1) is requested. This is because we found

more method implementation instances per developer than method usage instances when we collected expertise data for Top-1 recommendation.

This analysis answers our first research question, **RQ1**. The question is, ***"Does the combined expertise model outperform the individual expertise models?"***. The answer is ***"No"***. As we can see from our generalized results in Table 5.4, regardless of the number of recommended expert developers, the individual expertise models always perform better than the combined expertise model. Nevertheless, looking at the accuracy percentages for the combined expertise model in Table 5.4, we can say that the combined expertise model can be considered as an equally significant model and alternative choice of our individual expertise models.

Furthermore, if we look at the average accuracies project-wise separately in Tables 5.1, 5.2, and 5.3, we can see that for Top-1 recommendations, the accuracies differ from more than 10% (for *Guava* and *Commons Collections*, or *Commons Collections* and *AspectJ*) to more than 25% (for *Guava* and *AspectJ*) for both of the individual and the combined expertise models. The differences in accuracy percentages are significantly reduced between *Guava* and *Commons Collections* for Top-5 and Top-10 recommendations (0.16% - 8.42%). In other words, *Guava* and *Commons Collections* perform almost the same when Top-5 and Top-10 expert developers are recommended. However, *AspectJ* does not follow this trend. Moving from Top-1 to Top-5 or Top-10 recommendations shows a greater overall improvement for *AspectJ* than for the other two projects. As a result, the relative difference between *AspectJ* and the other two projects remains substantial (8.79% - 20.46%). This implies that the relative improvement between Top-1 and Top-5 or Top-10 recommendations for *Guava* and *Commons Collections* is smaller than the relative improvement between Top-1 and Top-5 or Top-10 recommendations for *AspectJ*.

Moreover, regardless of the number of recommended expert developers, *AspectJ* provides the highest accuracy among all three dataset projects for both of the individual and the combined expertise models. This effect occurs due to the differences in team and project

size among the dataset projects. Recall that *AspectJ* has the smallest number of contributors (6) and the largest number of commits (8,130) in our dataset (see Table 4.1 on Page 28). Therefore, we can say that the bigger the project size (i.e., # of commits) and the smaller the development team size (i.e., # of contributors), the better the recommender system performs. This analysis answers our second research question, **RQ2**. The question is, ***"Does the project and development team size affect the overall results?"***. The answer is ***"Yes"***. As we can see, due to having the largest project size and the smallest development team size, regardless of the number of recommended expert developers and the expertise model, *AspectJ* performs better than the other two projects of our dataset.

The data presented in Table 5.1, 5.2, and 5.3 show that for the individual expertise models (except for some cases in Top-1 recommendations), the UE model performs better than the IE model. This is likely to happen due to the ratio of usage expertise information to implementation expertise information (i.e., a developer uses a method more often than s/he changes the implementation). As a consequence, when we determine expertise based on the count of usage or implementation by the developers, it is reasonable for the usage expertise to take the lead. At the same time, if we look at the results for the CE model, we observe almost similar accuracies, except for slightly better percentages at the decimal level for CE-4 with `WeightSet-4` (`75%` UE, and `25%` IE), which can simply be disregarded. This implies that the weightings of the original usage expertise and implementation expertise scores are trivial. This analysis answers our third research question, **RQ3**. The question is, ***"What combination of usage and implementation expertise produces better results?"***. Although all the weightings produce almost the same results, in order for answering this question, we can say ***"WeightSet-4"*** (see Table 4.2 on Page 35). This is also reasonable as `WeightSet-4` prioritizes UE more than IE, and we proved earlier while answering **RQ1** that the UE model has more significance than the IE model.

Average Commit-wise Accuracy Results

The average commit-wise accuracy results reflect the percentage of query commits in each project, for which successful recommendations are made. In other words, it shows the probability of getting successful recommendations for commits in a project. Tables 5.5, 5.6, and 5.7 show the $Commit_{acc}$ results for the dataset projects for Top-1, Top-5, and Top-10 recommendations, respectively.

Table 5.5: Average Commit-wise Accuracy Results (Top-1)

Project Name	**UE**	**IE**	**CE-2**	**CE-3**	**CE-4**
`guava`	65.93%	55.84%	67.26%	67.30%	67.92%
`collections`	72.36%	62.72%	71.77%	71.77%	71.77%
`aspectj`	85.60%	82.75%	85.07%	85.05%	86.03%

Table 5.6: Average Commit-wise Accuracy Results (Top-5)

Project Name	**UE**	**IE**	**CE-2**	**CE-3**	**CE-4**
`guava`	90.25%	73.84%	90.60%	90.59%	90.76%
`collections`	87.83%	70.86%	85.63%	85.63%	85.63%
`aspectj`	96.08%	90.92%	96.96%	96.96%	96.96%

Table 5.7: Average Commit-wise Accuracy Results (Top-10)

Project Name	**UE**	**IE**	**CE-2**	**CE-3**	**CE-4**
`guava`	92.12%	73.99%	91.70%	91.70%	91.70%
`collections`	90.01%	70.86%	87.16%	87.16%	87.16%
`aspectj`	96.12%	90.92%	97.00%	97.00%	97.00%

Analysis of Results

From the Tables 5.5, 5.6, and 5.7, we can see that the commit-wise results follow a similar pattern to the method-wise results except with higher accuracies. The accuracy increase is likely due to commits being comprised of multiple methods for which we are predicting expertise. In other words, as a query commit is said to be successful if an expert is found for at least one query method, the overall success rate becomes higher than the more granular method-wise calculations. We achieved the lowest average accuracy of `55.84%` with implementation expertise in project *Guava* for Top-1 recommendation, and the highest average accuracy of `97%` with combined expertise in project *AspectJ* for Top-10.

Similar to the method-wise accuracies, the usage expertise performs better than the other expertise types almost always (Table 5.8). Similarly, *AspectJ* produces better average accuracies than the other two projects in commit-wise calculations as well. Moreover, among the weightings of the combined expertise model, `WeightSet-4` produces slightly better results in the decimal level. These statements further support the answers to our research questions **RQ1**, **RQ2**, and **RQ3**, which we previously discussed.

Table 5.8: Average Commit-wise Accuracy Results Across All Projects

Expertise Type	Top-1	Top-5	Top-10
UE	74.63%	91.39%	92.75%
IE	67.10%	78.54%	78.59%
CE-2 (`25%` UE, `75%` IE)	74.70%	91.06%	91.95%
CE-3 (`50%` UE, `50%` IE)	74.71%	91.06%	91.95%
CE-4 (`75%` UE, `25%` IE)	75.24%	91.12%	91.95%

Expertise Recommendation Results Over Time

For plotting the graphs to examine the accuracy results over the project lifetime, we calculated $Method_{acc}$ and $Commit_{acc}$ at each commit for the dataset projects, starting from

the initial commit to the latest commit of our dataset. At each query commit point, we used the data before that commit to calculate the accuracies. We have plotted the query commit serial numbers on the `x-axis`, while the method-wise and commit-wise average accuracy percentages for each query commit on the `y-axis`. Recall that the commits are ordered chronologically and numbered backwards in our dataset. In other words, in a project with `N` number of commits, `0` denotes the latest commit and `N-1` denotes the oldest one. Therefore, the `x-axis` of our graphs shows the commit serial numbers in a reverse order.

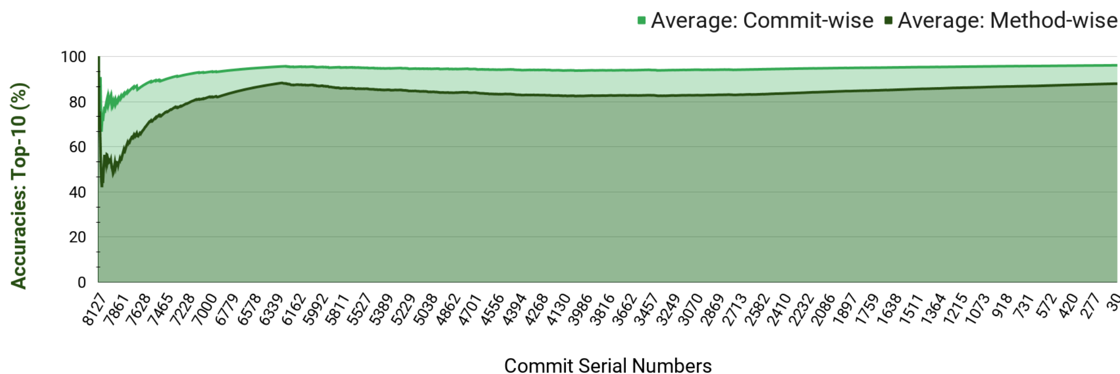


Figure 5.1: Average accuracy graph of Usage Expertise data for Top-10 (*AspectJ*)

Figure 5.1 shows graphs of the method-wise and commit-wise average accuracies for project *AspectJ* for Top-10 recommendation. The chart area shows the average accuracy change throughout the project's version history, starting from the initial commit (commit date: December 16, 2002), to the latest commit of our dataset (commit date: May 1, 2020). As discussed in Subsection 5.1.1, we can see that both of the graphs follow a similar pattern, but with higher accuracy values for the commit-wise data. We can see that at the initial commit, the accuracies were `100%`, which means the commit-author for that query commit was found within the Top-10 recommendation for the query methods within that commit. This happens because at that point in time there is only one developer who has contributed to the project. At the start of a project, even though there are multiple project members, they can only contribute or add files to the project sequentially. Over time, the contributions

of the other project members are added and appear in the repository. As soon as the contributors increase, the recommendations start to become less successful, and as a result, we can see a steep decline in the graphs. However, the decline is also for a short period here, as after around 500 commits, we see the graphs rising again. This is likely when the new developers have made enough contributions to be recommended, and the recommendations become successful.

Additionally, we can observe from the graphs that after nearly 1900 or approximately 25% commits (at commit #6,253), substantially, the graphs start maintaining the same level of accuracy and reach a plateau. A similar pattern is observed in all the graphs for all of the dataset projects (see Appendix B), regardless of the combinations of expertise types. With this information, we can understand how much training data the recommender system requires to provide highly accurate recommendations.

5.1.2 Mean Reciprocal Rank (MRR)

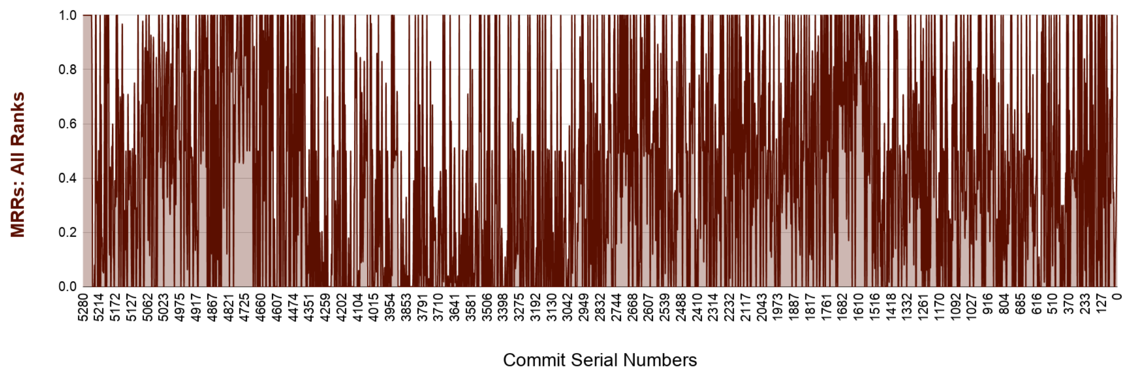


Figure 5.2: MRR chart with Implementation Expertise data for project *Guava*

We calculated the Mean Reciprocal Rank (MRR) for the query commits of our dataset projects using Equation 4.4 and plotted graphs of the results. However, we found that neither the results nor the charts provided meaningful data to evaluate the recommender system. Figure 5.2 shows an example of one such chart. This chart was constructed using

the data for each query commit of the dataset project *Guava*. It shows the implementation expertise data for its developers. Since MRR is about where in the rankings the correct answer tends to, no rank limits are set here (i.e., all ranks are considered).

As we can see from Figure 5.2, no decision can be made about the recommender system's performance using this metric for commit-based recommendations. Instead, we investigated a method-based MRR. For the Top-15 most frequently used and implemented methods of each dataset project, we calculated MRR_{method} according to Equation 4.5. Figure 5.3 shows the bar charts constructed with the MRR data from method-based calculations for projects *Guava*, *Commons Collections*, and *AspectJ*. Due to time constraints, we conducted the experiment for the Combined Expertise Model with `WeightSet-3` (`50%` UE and `50%` IE) only.

The `x-axis` in Figure 5.3 represent the MRR values for the Top-15 project methods. We excluded all of the most frequently used library methods while generating data for the charts. The exclusion of library methods was important here, as we generated data for the CE model only. To be precise, the CE model includes data from both of the UE model and the IE model, and we do not have any implementation information for the library methods in our dataset. Also, we excluded all the incorrect recommendations (i.e., the results for which we found *rank = 0*). The bars with MRR value `1` represents the methods for which the expert developer is recommended at rank `1`. Therefore, it means, the closer the MRR values to `1`, the better the recommendations provided.

From the charts, we can see that for *AspectJ*, almost `75%` of the methods received the correct recommendation at rank `1`, and the rest of the methods had a correct recommendation close to rank `1`. On the other hand, for *Guava*, only `20%` and for *Commons Collections*, almost `50%` of the methods received correct recommendation at rank `1`. If we look at Table 5.1 again, we can see that despite consisting of a small dataset, our MRR findings resemble our accuracy findings for CE-3. The MRR results also support the answer to **RQ2**.

Figure 5.3: Method-wise MRR graphs with Combined Expertise data at WeightSet-3 (`0.5, 0.5`) for projects (a) *Guava*, (b) *Commons-Collections*, and (c) *AspectJ*

5.2 Comparison to Previous Approach

We compare our work to the work of Ma et al. [7] using an evaluation approach similar to that which they used. The original authors evaluated their expertise models using the following heuristics:

Usage Expertise Heuristics: Depth of Method Knowledge, Breadth of Method Knowledge, Relative Depth of Method Knowledge, and Relative Breadth of Method Knowledge.

Implementation Expertise Heuristics: Method Change Frequency and Most Recent Change(s).

In this thesis, we examined Depth of Method Knowledge (Usage Expertise) and Method Change Frequency (Implementation Expertise) only. Initially, we planned to experiment with all of the heuristics the original authors experimented with. However, because of time constraints, we chose to focus on these two only, as these two are the closest to what we are looking into in our thesis. Examination using the remaining four heuristics is left as a future work (see Section 6.1).

5.2.1 Depth of Method Knowledge

Determining the depth of a developer's knowledge on a method involves quantifying the frequency of calls made by that developer to that method. According to this heuristic, the developer who has made the most calls to a method is inferred to be the developer with the most expertise for that method. This corresponds to what we refer to as usage expertise (see Subsection 3.1.1). To get the results of applying this heuristic on our combined model, we used our overall expertise calculating Equation 3.1.

5.2.2 Method Change Frequency

Ma et al. [7] used the method change frequency heuristic to measure the frequency of methods implemented by a developer. It is equivalent to our implementation expertise (see Subsection 3.1.2). According to this heuristic, the developer who has made the most changes to a method is inferred to be the developer with the most expertise for that method.

To get the results of applying this heuristic on our combined model, we used our overall expertise calculating Equation 3.1.

5.2.3 Performance Analysis

Ma et al. [7] experimented with the *AspectJ* and *Eclipse* projects to evaluate their recommendation system. Of these two, we used *AspectJ* to evaluate our recommendation system. Therefore, we use the results for *AspectJ* to compare both of the approaches.

Figure 5.4 shows the performance graphs of the recommender system by Ma et al. [7] for project *AspectJ* for Depth of Method Knowledge and Method Change Frequency (the dashed lines)[19]. The figure also includes the performance of our recommender system for project *AspectJ* for Depth of Method Knowledge (i.e., Usage Expertise), Method Change Frequency or (i.e., Implementation Expertise), and Combined Expertise at `WeightSet-3` (the solid lines). We show the chart data for only one `WeightSet` here, as the results were found to be nearly the same for all of the weightings.

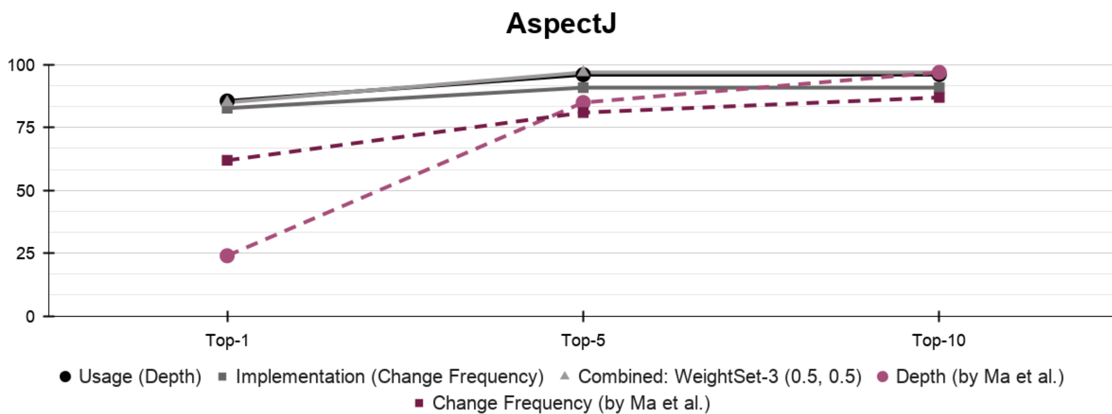


Figure 5.4: Performance analysis graphs for both of the approaches for project *AspectJ*

Ma et al. [7] considered the whole method set in a query commit as input to their system. This is equivalent to our commit-wise average accuracy results (Tables 5.5, 5.6, and 5.7). Comparing the graphs, we can see, when Top-1 recommendations are considered, our ap-

[19]The data point values are estimated from the provided graphs in Ma et al. [7] in Figure 3.

proach performs more than three times better than the original authors' approach for Depth of Method Knowledge and more than 30% better for Change Frequency. In other words, our approach results in the correct developer being the top recommendation more often. Even though their Depth of Method Knowledge and Method Change Frequency are equivalent to our Usage Expertise and Implementation Expertise, respectively, our results are better because we refined our query methods and results in several steps (see Subsection 4.2.5). Moreover, the most important feature of our recommender system is that we resolved most of the query method signatures completely. This provided us with more accurate details about each method and helped our system to allocate the developers' expertise information to the proper methods. On the contrary, Ma et al. [7] did not resolve the query methods. As a result, their recommender system works based on method names only. For example, in Figure 5.3, in the Top-15 methods for *Guava*, we can see four methods with the name `add`. These four are different methods with different origins. Our system distinguishes those and generates separate results for each of those. Conversely, Ma et al.'s [7] recommender system would consider those four methods as a single method because of having the same name and generate only one recommendation.

Similar to Top-1, when Top-5 recommendations are considered, our methodology performs more than 10% better than the original authors' approach for both of the expertise types. However, when Top-10 results are considered, both of the approaches perform almost the same and provide nearly 100% accuracy. This result happens because the number of contributors for *AspectJ* is less than 10. As a result, while considering Top-10 developers for recommendations, the expected developer (i.e., commit author) is always[20] recommended. Interestingly, when we compare Ma et al.'s [7] results with our Combined Expertise results (CE-3 results in Figure 5.4), we find that our CE model performs better for all of the cases, except for Depth of Method Knowledge for Top-10, where both of the approaches perform the same (97%).

[20]Except for cases where the commit author is a new developer.

From this analysis, we can say that our recommender creation approach outperforms that of Ma et al. [7] including our combined expertise model, which represents a combination of Ma et al.'s [7] usage expertise and implementation expertise models. This answers our fourth and the final research question, **RQ4**. The question is, ***"Do our individual and combined expertise models outperform the previous approach by Ma et al. [7]?"***. The answer is ***"Yes"***. Nevertheless, this statement is correct according to the results of one common dataset (*AspectJ*) between the two projects. Future experimentation is needed to determine if this result generalizes to other project, such as *Eclipse*.

5.3 Threats to Validity

In this section, we discuss threats to the construct, external, and internal validity of our results.

5.3.1 Construct Validity

Our approach requires measuring developers' method expertise to construct the recommender. We are collecting artifacts in the form of developers' method usage and implementation and using these artifacts as expertise units. Hence, we believe that the threats to construct validity are minimal for our results.

5.3.2 External Validity

We evaluated the effectiveness our recommendation system with three different types of dataset projects of sizes large, medium, and small. Further, we used the complete dataset for each of the projects (i.e., we considered all the commits for the projects throughout their lifetime). Our recommender creation approach does not depend on the size or the type of project. Even though we restricted our investigation to `Java` projects using the `Git` version control system, our approach is independent of programming language and version control system. Therefore, we believe that the threat to generalizability is minimized, although investigation with more systems would further reduce this threat.

5.3.3 Internal Validity

Resolution of Methods

Repetitive data: We used the `JavaParser`[21] library in our recommender system to parse the `Java` source code. The parser resolves Method Declarations and Method Invocations differently. The MethodDeclarationResolver resolves the names only for the parameter and the return object types. On the other hand, the MethodInvocationResolver resolves the complete signature for the parameter and the return object types. This situation sometimes results in the repetition of the same method. For example, the method from project *Apache Commons Collections* named `getTypeTransformer` of class `BeanMap` under package `org.apache.commons.collections` has a parameter of `Class` type and returns an object of `Transformer` type. The MethodDeclarationResolver resolves the method as:

- `org.apache.commons.collections.BeanMap.getTypeTransformer`
  `- parameter(s) ([Class])`
  `- returns [Transformer]`

where, the MethodInvocationResolver resolves the method as:

- `org.apache.commons.collections.BeanMap.getTypeTransformer`
  `- parameter(s) ([java.lang.Class])`
  `- returns [org.apache.commons.collections.Transformer]`

To eliminate the repetitive data, we split the resolved return types and the parameter types to extract the class names only. However, we cannot confirm that the threat is eradicated for all of the methods. As a consequence of having multiple instances of the same method, there is a possibility that the results may contain some false negatives.

Matching method names: Recall that we followed some steps to address the exceptions that occurred while resolving method invocations (see Subsection 4.2.1). All of the measures taken to resolve the methods that the parser library failed to do, follow a method-name

[21] https://github.com/javaparser/javaparser

matching strategy. We cannot claim that this strategy provides accurate results all the time. In other words, because of method overloading and method overriding, the name matching strategy can result in false answers.

Unresolved methods: While investigating the unsuccessful or incorrect recommendations by the system (Subsection 4.2.5), recall that we filter out the methods from our query method list that are unresolved by the system. We recognize that this could be a potential threat, but we believe it does not invalidate our results significantly. As the occurrence of unresolved methods by the system is negligible (only 6%), we are confident that our results are accurate. What proportion of unresolved methods will result in the recommender not being feasible was unexplored.

Measuring Implementation Expertise

As explained earlier, we calculate the implementation expertise of a developer using the frequency of insertions made by the developer on a method (Subsection 3.1.2). The insertions may include blank lines and comments, with or without actual code statements. This may contribute to some false positives in the results. Additionally, the following two snippets of code are functionally equivalent, yet one would be counted as four insertions (*Snippet-1*), and another would be counted as one insertion (*Snippet-2*). However, that would not affect the results, as we are looking at a broader context, that is, as long as the coding style is consistent within the project, the counting will be consistent.

Snippet-1	***Snippet-2***

`if (x == 6)`
`{`
`    x++;`
`}`
	`if (x == 6) {x++;}`

Developer Identification

The developer list for a project is constructed using the unique combination of name and email address. This implies that the developers with the same names but different email addresses are considered as different persons. Similarly, the developers with the same email addresses but different names are considered as different persons. For example, a developer from project *Eclipse AspectJ* named *Andy Clement* appears thrice in our system with the same username *Andy Clement*, but with three different email addresses – *aclement@gopivotal.com*, *aclement@pivotal.io* and *aclement@vmware.com*. Similarly, a developer from project *Apache Commons Collections* named *Alex Herbert* appears twice in our system with the same email address *a.herbert@sussex.ac.uk*, but with two different usernames – *aherbert* and *Alex Herbert*.

One reason for such cases could be when the same developer possesses multiple usernames and email addresses because of accessing the version control system from multiple devices. As a result of such conflicts, a recommendation list can have multiple entries for developers with the same name (see Figure 4.2 on Page 34) because of having different email addresses. Similarly, a recommendation list can have multiple entries for the same developer because of having different names. This situation could generate some false negatives in the results, as separate entries for the same person splits their contributions and impacts the ranking. To reduce the effect of this problem, we compare the names of the commit author and the recommended developer at first, and then the email addresses, while evaluating a recommendation. This approach merges the developers with the same names or the same email addresses, but at the same time, could generate false positives in the results. In other words, in either way, we cannot avoid false results without having direct contact with the developers to obtain their proper identity. However, there are not many instances of such cases in our dataset. We found that only `3.18%` of the developers in our dataset have different email addresses with the same usernames. On the other hand, only `4.11%` of the developers in our dataset have different usernames with the same email ad-

dresses. With such a low amount of conflicting data, we believe this threat is trivial for the validity of our results.

5.4 Summary

In this chapter, we have presented the results of evaluating our recommender system using three different dataset projects according to the evaluation strategy presented in Chapter 4. Our experiments show excellent results, regardless of the software development projects' team size and domain. We found that our approach resulted in almost 90% average accuracy for Top-10 recommendations. Moreover, after comparing our results with that of Ma et al.'s [7] (the authors of the research on which this work is based), we found that we achieved better accuracy.

While analyzing our results, we found the answers to the research questions of this thesis. Our first research question was, ***Does the combined expertise model outperform the individual expertise models?***. The answer is ***"No"***. As we found that regardless of the number of recommended expert developers, the individual expertise models always perform better than the combined expertise model. Our second research question was, ***Does the project and development team size affect the overall results?***. The answer is ***"Yes"***. As we found that due to having the longest history and the smallest development team size, regardless of the number of recommended expert developers and the expertise model, *AspectJ* performed better than the other projects of our dataset. The third research question was, ***What combination of usage and implementation expertise produces better results?***. Although all the weightings produce nearly the same results, for answering this question, we chose ***"WeightSet-4"***. Finally, our fourth research question was, ***Do our individual and combined expertise models outperform the previous approach by Ma et al. [7]?***. After comparing the results of one common dataset (*AspectJ*) between the two projects, we found that the answer is ***"Yes"***. In the end, we discussed the situations that could be threats to the validity of our results and the steps that we took to minimize the threats.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented an expert developer recommender system for software project methods. Our approach used method usage and implementation expertise for developers of a particular software project to measure a developer's expertise.

Our goal with this work is to help establish better communication between software project developers, as well as the team manager and the developers. When a new developer joins a team, or a developer is assigned a task that s/he is completely unfamiliar with, or a team manager is looking for the appropriate developer to assign a feature, a large amount of time is spent finding the correct person for help. Effective communication and coordination among software project personnel have always been an issue, which is increasing every day with the global and distributed nature of software development. Our recommender system addresses these situations and suggests the relevant expertise to approach, which saves valuable production time.

To create the expertise recommender, we mine the version control repository and the source code repository of a project to extract the developers' contribution to the project over its lifetime. We collect artifacts in the form of developers' method usage and implementation as expertise atoms. After finding the usage and implementation contributions of the developers for the methods of a project, we combine the values for the two types of expertise to determine the developers' overall method expertise. According to our evaluation strategy (adopted from Ma et al. [7]), we consider a recommendation successful, when the actual commit author of a method's usage or implementation is listed within the Top-N

recommendation for that method.

We have used three software dataset projects of different sizes and from different organizations to validate the system. Our chosen projects are *Google's Guava*, *Apache Commons Collections*, and *Eclipse's AspectJ*. We computed accuracy and Mean Reciprocal Rank with the results generated by the system for the dataset projects.

Our experiments show excellent results, regardless of software development projects' team size and domain. We found that our approach resulted in almost 90% average accuracy for Top-10 recommendations. Moreover, after comparing our results with that of Ma et al.'s [7] (the authors of the research on which this work is based), we found that we achieved better accuracy. Through this work, we found answers to the following research questions:

RQ1: Does our combined expertise model outperform our individual expertise models?

The answer is **"No"**. We found that regardless of the number of recommended expert developers, the individual expertise models always perform better than the combined expertise model.

RQ2: Does the project and development team size affect the overall results?

The answer is **"Yes"**. We found that due to having the longest history and the smallest development team size, regardless of the number of recommended expert developers and the expertise model, *AspectJ* performed better than the other two projects of our dataset.

RQ3: What combination of usage and implementation expertise produces better results?

Although all the weightings produce almost the same results, we found **"WeightSet-4"** (i.e., 75% usage expertise and 25% implementation expertise) to be the best combination.

RQ4: Do our individual and combined expertise models outperform the previous approach by Ma et al. [7]?

The answer is **"Yes"**. However, this statement is correct according to the results of

one common dataset (*AspectJ*) between the two projects. Future experimentation is needed to determine if this result generalizes to other project, such as *Eclipse*.

To summarize, we successfully created a method expert developer recommendation system. We explored three different models for the recommender, a usage expertise model, an implementation expertise model, and an overall expertise model using the combination of usage expertise and implementation expertise of the developers. Through several experiments, we found that the usage expertise model performs the best to recommend expert developers for a method. The overall expertise model also performs well, which makes it an excellent alternative to the usage expertise model. However, we found that the implementation expertise information performed the worst. Yet has importance while determining the combined expertise nonetheless.

6.1 Future Work

There are a number of ways in which this work could be improved and extended. The following are ideas for future work.

6.1.1 Cleanup repetitive data

As discussed in Subsection 5.3.3, the parser library we used generates some repetitive data. Although we have taken some measures to clean those up, there remains such data in the system. In the future, we plan to conduct manual inspections to identify repetitive entries for the same method, to ensure better accuracy.

6.1.2 Filter comments and blank lines

While determining implementation expertise for the developers, we did not exclude the comments and the blank lines (Subsection 5.3.3). These could be filtered out to reduce the number of false positives, if this causes any.

6.1.3 Work with deletions

In our approach, we only consider the insertions made by the developers to assess their expertise. To examine the expertise required to remove or delete source code, we plan to explore the deletions made by the developers along with the insertions. Through this experiment, we will be able to find out if the deletions should be considered while computing the method implementation expertise of developers.

6.1.4 Add support for other programming languages

Even though we restricted our investigation to `Java` projects using the `Git` version control system, our approach is independent of programming language and version control system. We will add necessary support for other programming languages and version control systems to our recommender system in the future.

6.1.5 Categorize methods and developer skills

As we have all the information about which developer works on which method and how frequently, we can use this data to categorize the methods and the skills of the developers of a project. This way, we can create skill-based profiles for the developers. These profiles can be useful not only for an existing system but also while allocating tasks to the developers for a new module or even a new project. Besides, the team managers can benefit while hiring developers from other companies with the help of skill-based profiles.

6.1.6 Experiment with more heuristics

Recall that due to time constraints, we could not experiment with all of the six heuristics Ma et al. [7] examined (Section 5.2). We chose only the two that are closest to our expertise models. In the future, we will explore the four remaining heuristics. This experiment will help us to find out if our approach outperforms that of Ma et al.'s [7] in all of their heuristics.

Bibliography

[1] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 1–8, Minneapolis, MN, USA, May 2007. IEEE.

[2] Apache Commons. Apache commons collections. `https://commons.apache.org/proper/commons-collections/`, July 2019. [Online; accessed 26-July-2020].

[3] AspectJ Developers. Eclipse aspectj. `https://projects.eclipse.org/projects/tools.aspectj`, July 2020. [Online; accessed 26-July-2020].

[4] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, Szeged, Hungary, September 2011. ACM.

[5] Colin Decker, James Sexton, and Joshua O'Madadhain. Guava wiki. `https://github.com/google/guava/wiki`, October 2016. [Online; accessed 26-July-2020].

[6] Huzefa Kagdi, Maen Hammad, and Jonathan I. Maletic. Who can help me with this source code change? In *2008 IEEE International Conference on Software Maintenance*, pages 157–166, Beijing, China, October 2008. IEEE.

[7] David Ma, David Schuler, Thomas Zimmermann, and Jonathan Sillito. Expert recommendation with usage expertise. In *2009 IEEE International Conference on Software Maintenance*, pages 535–538, Edmonton, AB, Canada, October 2009. IEEE.

[8] David W. McDonald and Mark S. Ackerman. Expertise recommender: A flexible recommendation system and architecture. In *Proceedings of ACM Conference on Computer Supported Collaborative Work*, pages 231–240, Philadelphia, Pennsylvania, USA, 2000. ACM.

[9] 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, Orlando, Florida, May 2002. ACM.

[10] 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, Leipzig, Germany, May 2008. ACM.

[11] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited; Analyse, transform and generate your Java code base*. Learnpub, https://leanpub.com/javaparservisited, May 2019. [This book is 50% complete].

[12] Igor Steinmacher, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Recommending mentors to software project newcomers. In *Third International Workshop on Recommendation Systems for Software Engineering*, pages 63–67, Zurich, Switzerland, July 2012. IEEE.

[13] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, Xavier Blanc, and Floréal Morandat. Automatic extraction of developer expertise. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, London, England, United Kingdom, May 2014. ACM.

[14] Wikipedia contributors. Changeset — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Changeset&oldid=796831103, 2017. [Online; accessed 10-July-2019].

[15] Wikipedia contributors. Version control — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Version_control&oldid=900636254, 2019. [Online; accessed 10-July-2019].

[16] Wikipedia contributors. Accuracy and precision — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Accuracy_and_precision&oldid=966550962, 2020. [Online; accessed 14-August-2020].

[17] Wikipedia contributors. Mean reciprocal rank — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Mean_reciprocal_rank&oldid=965530058, 2020. [Online; accessed 14-August-2020].

[18] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 513–523, Orlando, Florida, May 2002. ACM.

Appendix A

Recommendation Results

All the recommendation results presented here use `0.5` as the values of the weights w_1 and w_2 while calculating the overall expertise.

Scenario-1: Method Overloading

The following figure demonstrates how our recommender system handles 'method overloading'. Here, we have used the example of method `copy` of class `ByteStreams` under package `com.google.common.io` from project `google/guava`.

```
Please enter your project name: guava
Please enter the method name from project 'guava': copy
Please enter the class name of method 'copy': ByteStreams
Please enter the package name of class 'ByteStreams': com.google.common.io

More than one results found for this method. Please choose one from the following options:

 1. com.google.common.io.ByteStreams.copy(['InputSupplier<? extends InputStream>', 'OutputSupplier<? extends OutputStream>']) --> [returns long]
 2. com.google.common.io.ByteStreams.copy(['InputSupplier<? extends InputStream>', 'OutputStream']) --> [returns long]
 3. com.google.common.io.ByteStreams.copy(['InputStream', 'OutputStream']) --> [returns long]
 4. com.google.common.io.ByteStreams.copy(['ReadableByteChannel', 'WritableByteChannel']) --> [returns long]

Please enter the serial number of the method of your choice: 1

~~~~~~Successfully retrieved expert developers~~~~~~

Developer Name ::: Usage Expertise Score
-----------------------------------------

1. Chris Povirk ::: 10.0
2. Colin Decker ::: 2.5
3. guava.mirrorbot@gmail.com ::: 2.5
======================================

Developer Name ::: Implementation Expertise Score
--------------------------------------------------

1. kevinb@google.com ::: 10.0
2. Chris Povirk ::: 7.06
3. Colin Decker ::: 1.76
4. Kurt Kluever ::: 0.59
======================================

Developer Name ::: Overall Expertise Score
-------------------------------------------

1. Chris Povirk ::: 8.53
2. kevinb@google.com ::: 5.0
3. Colin Decker ::: 2.13
4. guava.mirrorbot@gmail.com ::: 1.25
5. Kurt Kluever ::: 0.295
======================================
```

Figure A.1: Recommendation results for method overloading

Scenario-2: No Usage Expertise Found

The following figure demonstrates an example of recommendations when no usage expertise contribution of any developer found for a method of a project. This example reflects that the recommended developers have created and modified the `createTempDir` method in class `Files` of project *Guava* but none of the developers of that project have ever used it anywhere.

```
Please enter your project name: guava
Please enter the method name from project 'guava': createTempDir
Please enter the class name of method 'createTempDir': Files
Please enter the package name of class 'Files': com.google.common.io

~~~~~~Successfully retrieved expert developers~~~~~~


Sorry, no usage expertise found for this method!

Developer Name ::: Implementation Expertise Score
--------------------------------------------------

1. zhenghua ::: 10.0
2. kevinb@google.com ::: 7.0
3. cpovirk ::: 5.5
4. kak ::: 1.0
======================================

Developer Name ::: Overall Expertise Score
-------------------------------------------

1. zhenghua ::: 10.0
2. kevinb@google.com ::: 7.0
3. cpovirk ::: 5.5
4. kak ::: 1.0
======================================
```

Figure A.2: Recommendation results when usage expertise not found

Scenario-3: No Implementation Expertise Found

The following figure demonstrates an example of recommendations when no implementation expertise contribution of any developer found for a method of a project. In this example, we recommended expert developers from project `apache/commons-collections` for the `java.util` library method `get` from class `ArrayList`. As a consequence of `get` being a java library method, clearly the `apache/commons-collections` developers do not possess any implementation history for the method.

```
Please enter your project name: commons-collections
Please enter the method name from project 'commons-collections': get
Please enter the class name of method 'get': ArrayList
Please enter the package name of class 'ArrayList': java.util

~~~~~~Successfully retrieved expert developers~~~~~~

Developer Name ::: Usage Expertise Score
-----------------------------------------

1. Stephen Colebourne ::: 10.0
2. Thomas Neidhart ::: 8.89
3. Rodney Waldhoff ::: 6.67
4. Gary D. Gregory ::: 5.56
5. Geir Magnusson Jr ::: 2.22
6. Craig R. McClanahan ::: 2.22
7. Gary Gregory ::: 2.22
8. Jason van Zyl ::: 2.22
9. Henri Yandell ::: 2.22
10. James Strachan ::: 1.11
11. Michael Smith ::: 1.11
======================================


Sorry, no implementation expertise found for this method!

Developer Name ::: Overall Expertise Score
-------------------------------------------

1. Stephen Colebourne ::: 10.0
2. Thomas Neidhart ::: 8.89
3. Rodney Waldhoff ::: 6.67
4. Gary D. Gregory ::: 5.56
5. Geir Magnusson Jr ::: 2.22
6. Craig R. McClanahan ::: 2.22
7. Gary Gregory ::: 2.22
8. Jason van Zyl ::: 2.22
9. Henri Yandell ::: 2.22
10. James Strachan ::: 1.11
11. Michael Smith ::: 1.11
======================================
```

Figure A.3: Recommendation results when implementation expertise not found

Appendix B

Expertise Accuracy Graphs

All the method-wise and commit-wise average accuracy graphs generated for the three dataset projects are presented here. The data for the graphs are produced using all of the commits for each of the project (see Table 4.1). In the graphs, the projects are named as:

- **Guava** for *Google's Guava*
- **Collections** for *Apache Commons Collections*
- **AspectJ** for *Eclipse's AspectJ*

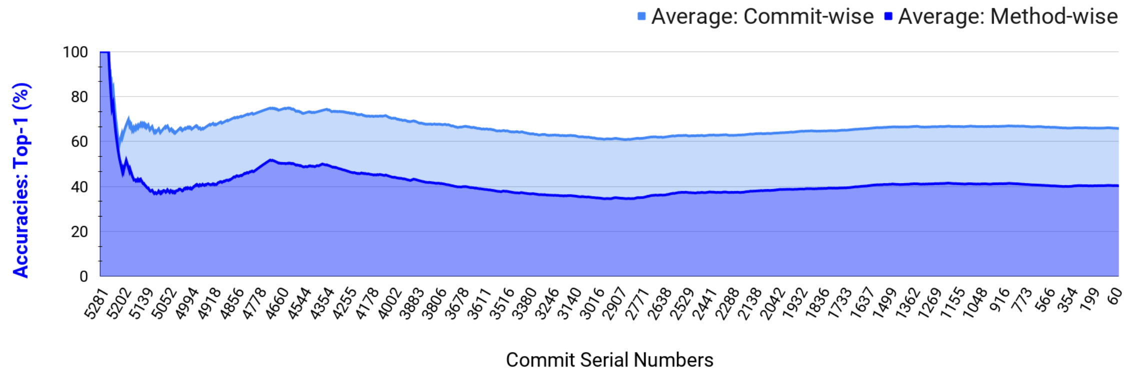
For each project, the graphs are presented in the following sequence:

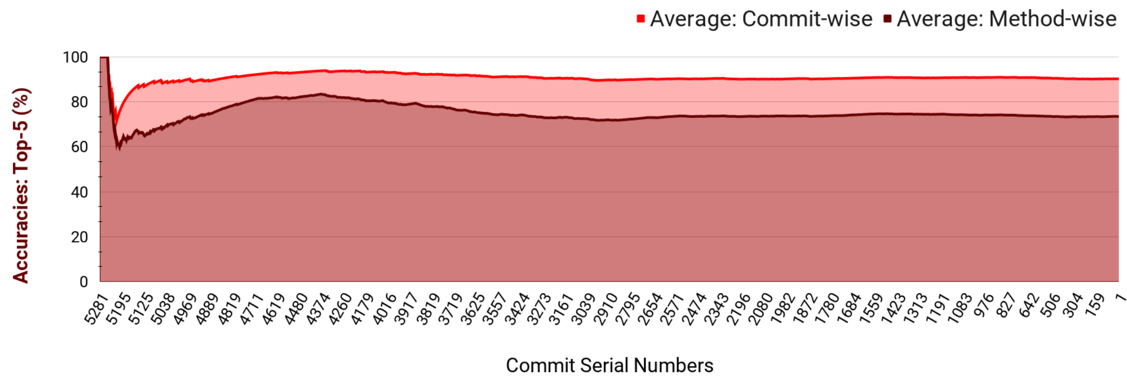
1. Graphs generated using the Usage Expertise data
2. Graphs generated using the Implementation Expertise data
3. Graphs generated using the Overall Expertise data at Weight Set - 1 ($w_1 = 0$ and $w_2 = 1$)
4. Graphs generated using the Overall Expertise data at Weight Set - 2 ($w_1 = 0.25$ and $w_2 = 0.75$)
5. Graphs generated using the Overall Expertise data at Weight Set - 3 ($w_1 = 0.50$ and $w_2 = 0.50$)
6. Graphs generated using the Overall Expertise data at Weight Set - 4 ($w_1 = 0.75$ and $w_2 = 0.25$)
7. Graphs generated using the Overall Expertise data at Weight Set - 5 ($w_1 = 1$ and $w_2 = 0$)

Each of the section mentioned above contains three graphs, for percentage of average accuracies at recommendations with (a) Top-1 expert developer (*theme: blue*), (b) Top-5 expert developers (*theme: red*), and (c) Top-10 expert developers (*theme: green*). The darker coloured areas in the graphs indicate the method-wise results, while the lighter coloured areas indicate the commit-wise results.

B.1 Google's Guava Graphs

B.1.1 Usage Expertise Graphs





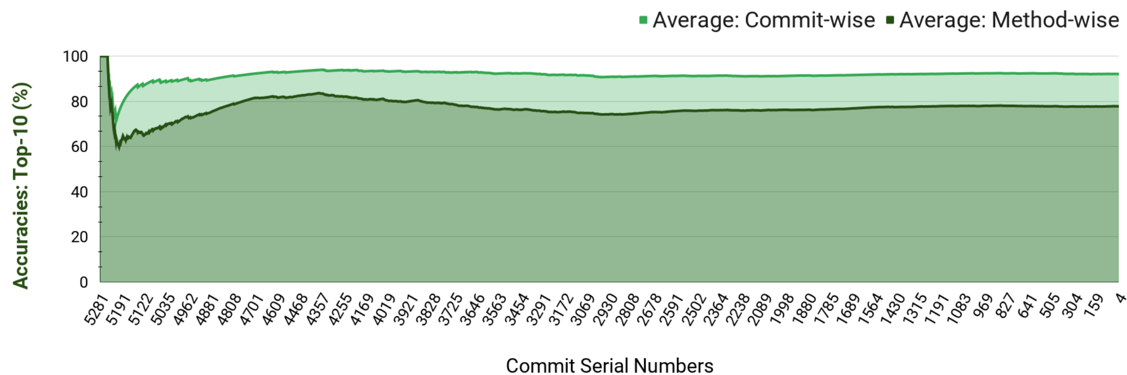
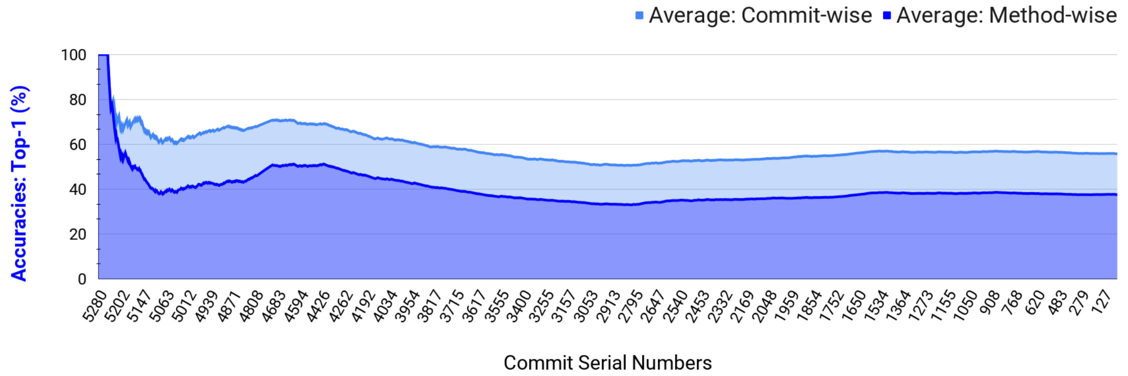
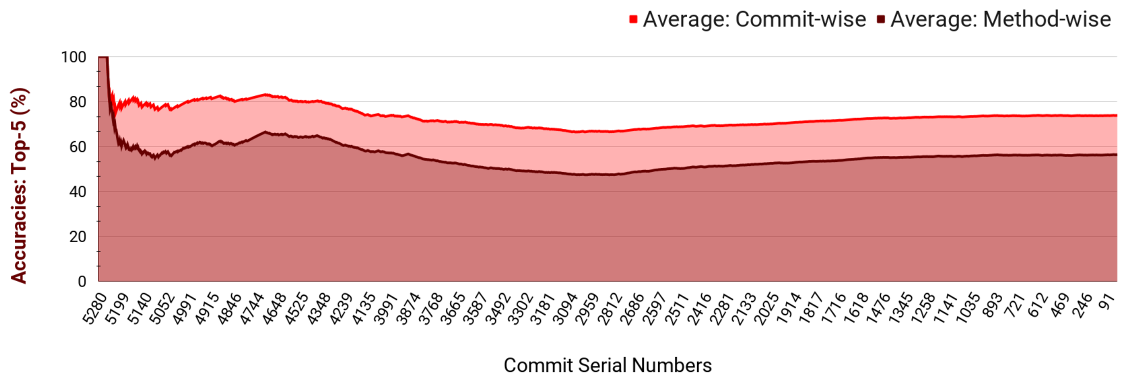


Figure B.1: Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Guava*

B.1.2 Implementation Expertise Graphs





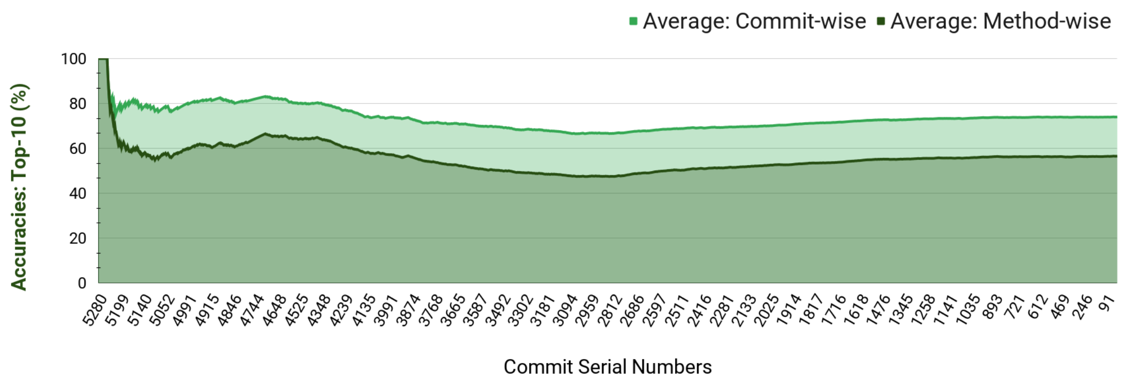
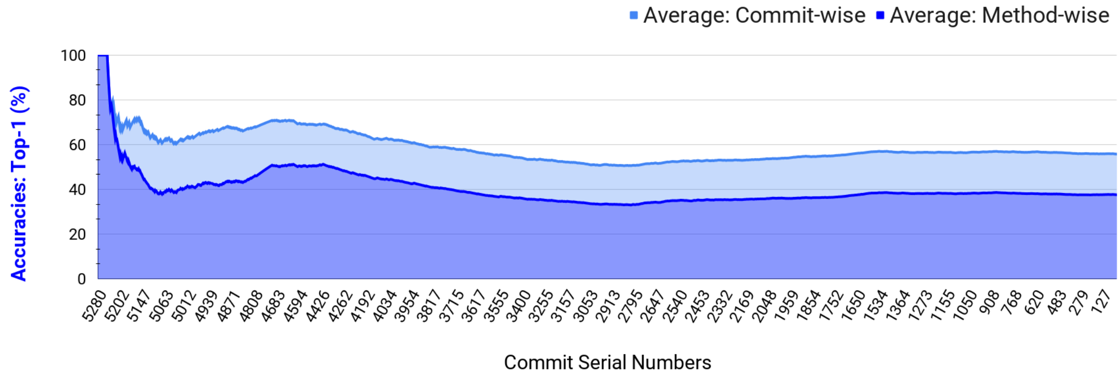
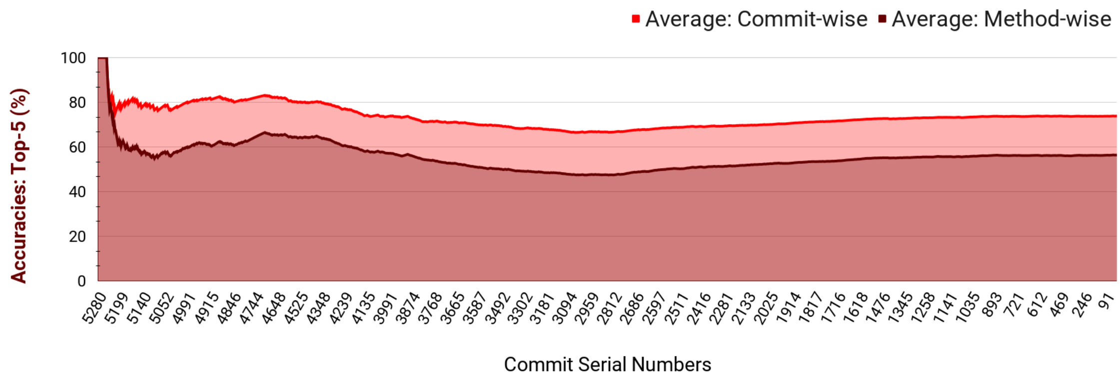


Figure B.2: Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Guava*

B.1.3 Combined Expertise Graphs

Weight Set - 1 ($\mathbf{w}_1$ = 0 and $\mathbf{w}_2$ = 1)





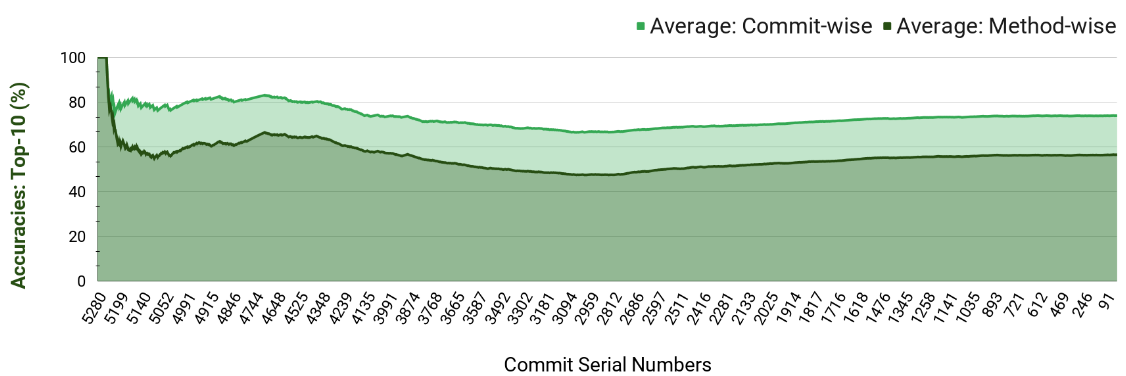
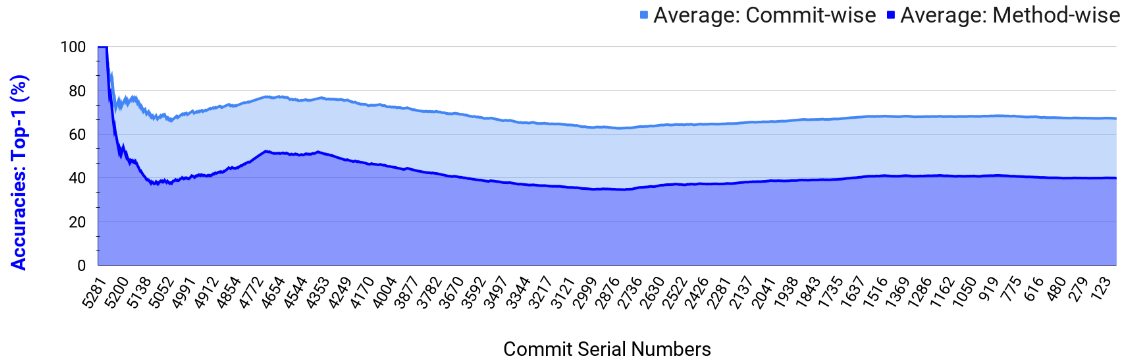
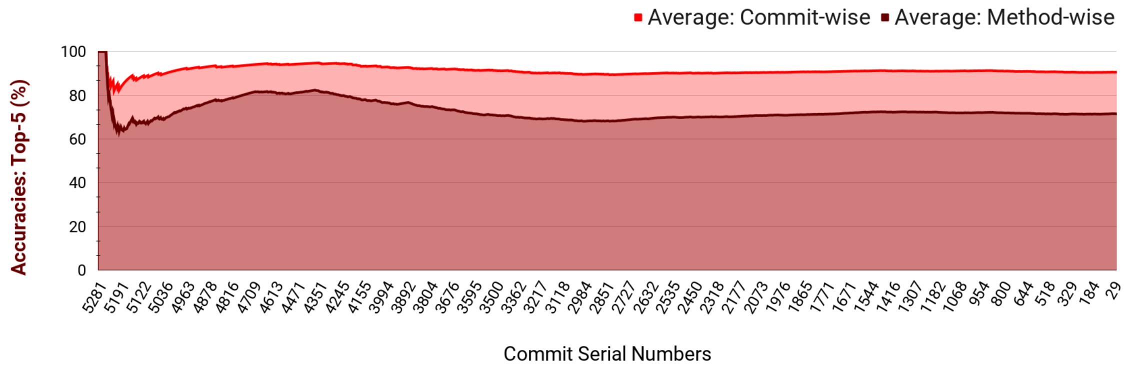


Figure B.3: Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *Guava*

Weight Set - 2 (w_1 `= 0.25` and w_2 `= 0.75`)





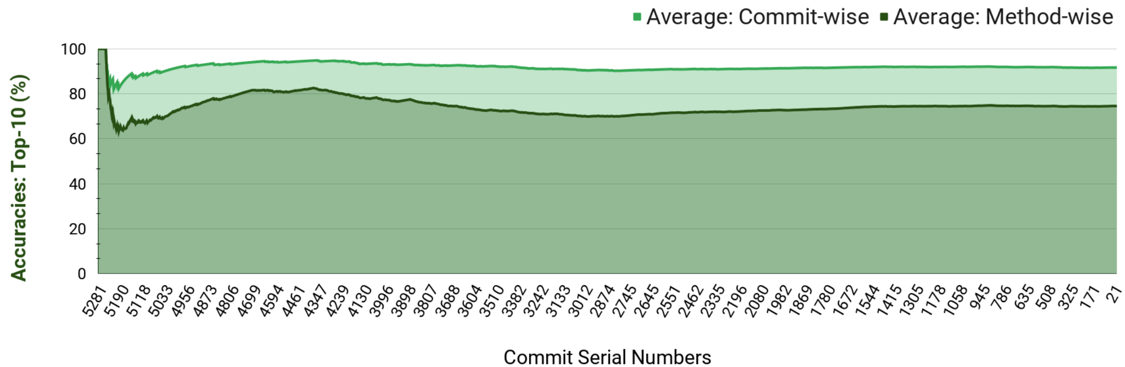
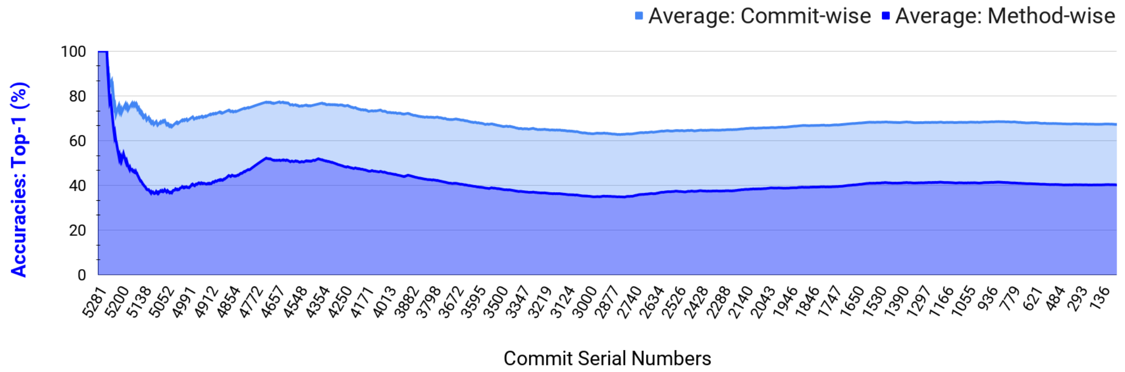
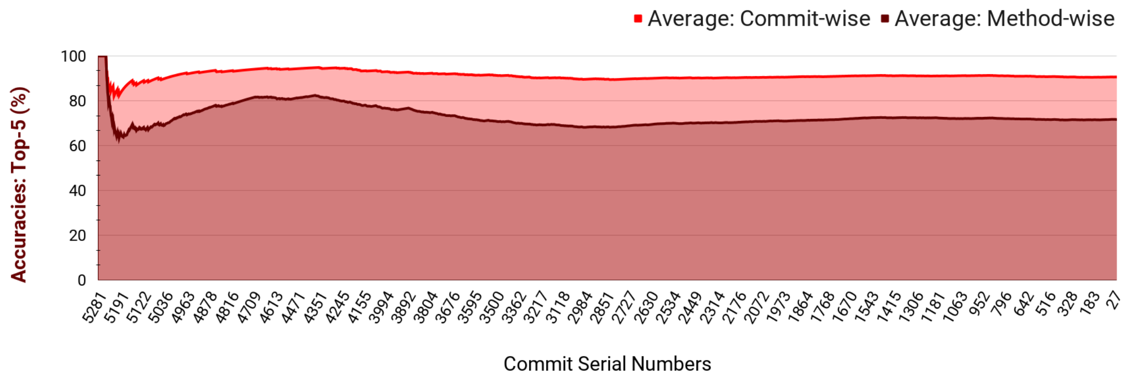


Figure B.4: Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *Guava*

Weight Set - 3 ($\mathbf{w}_1$ `= 0.50` and $\mathbf{w}_2$ `= 0.50`)





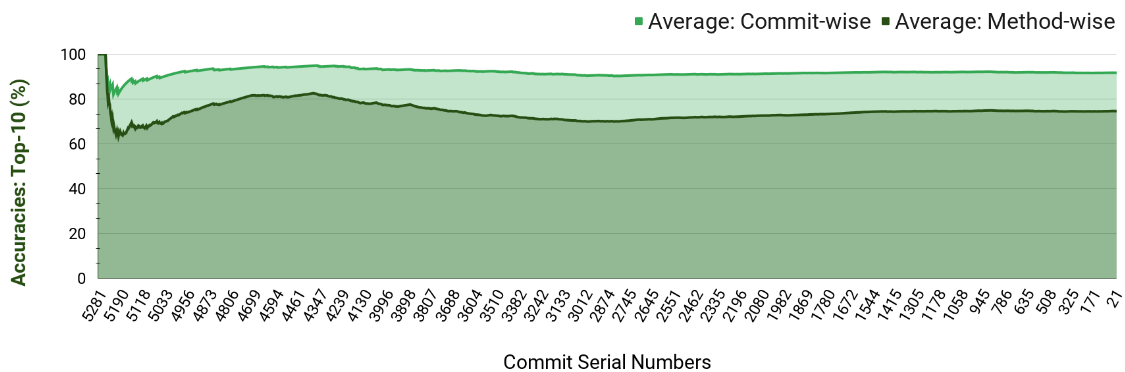
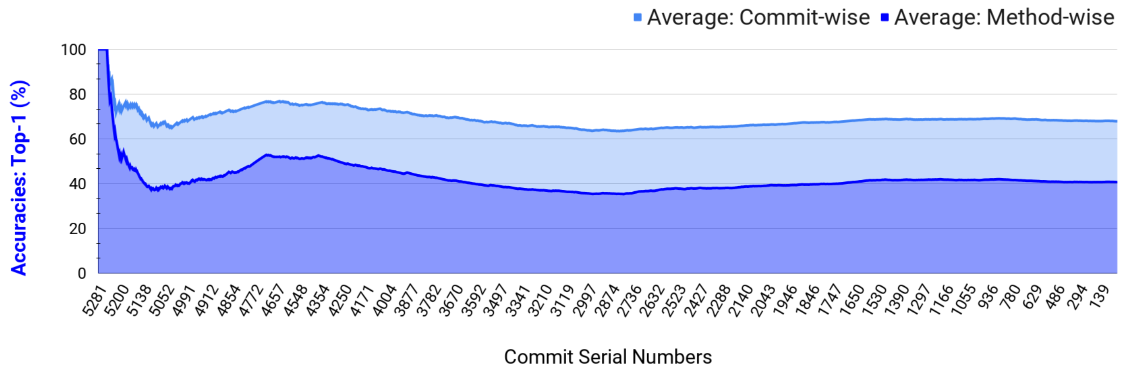
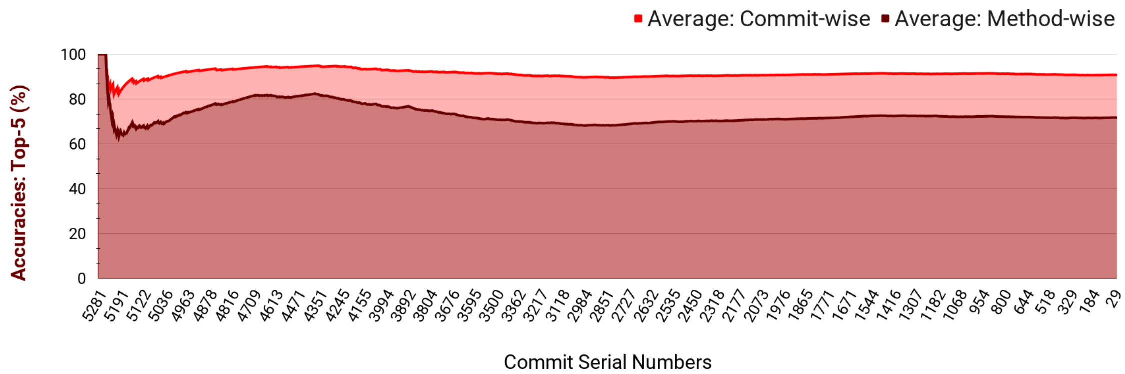


Figure B.5: Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *Guava*

Weight Set - 4 (w_1 = 0.75 and w_2 = 0.25)





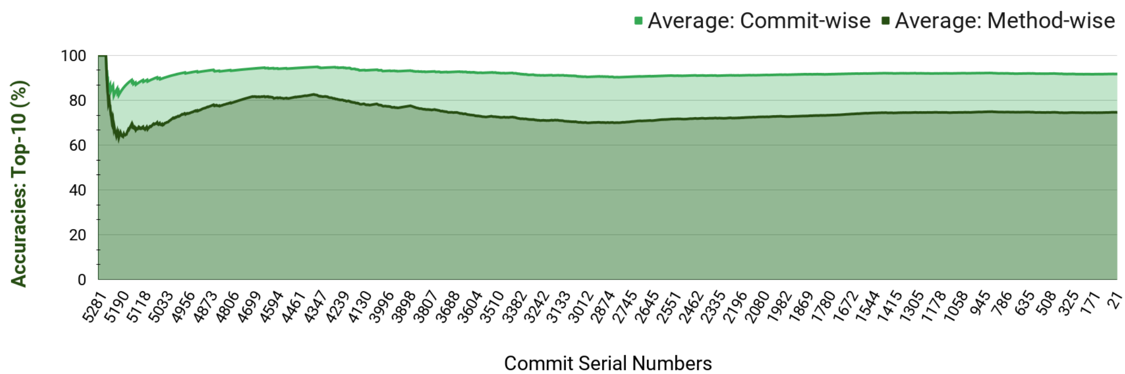
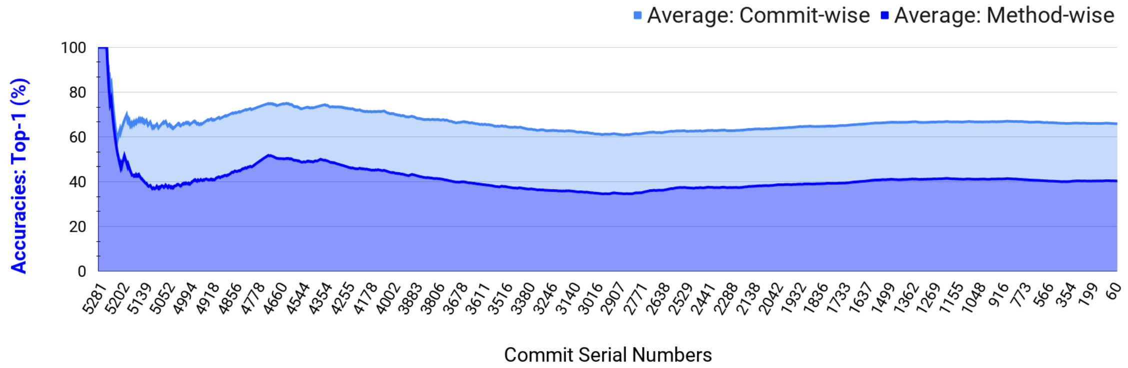
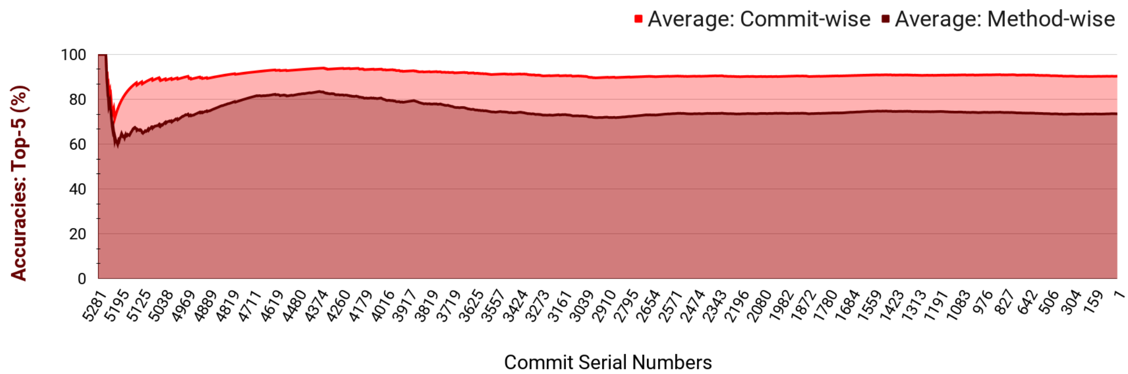


Figure B.6: Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *Guava*

Weight Set - 5 (w_1 = 1 and w_2 = 0)





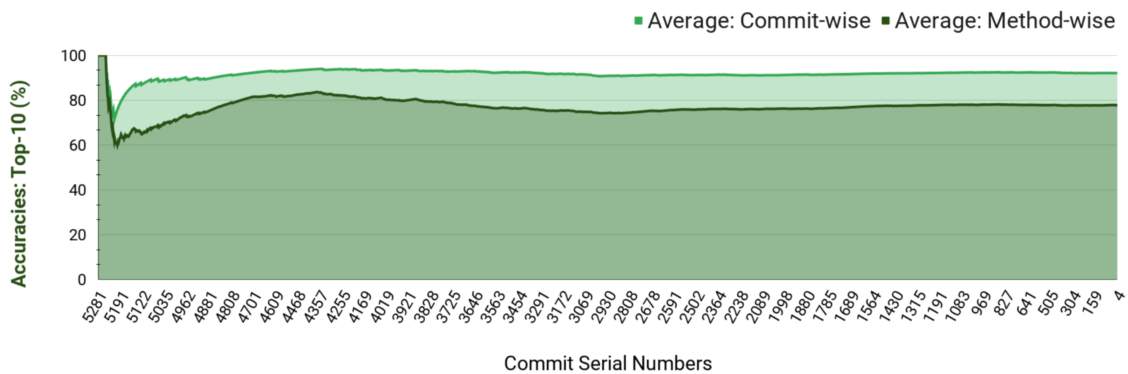
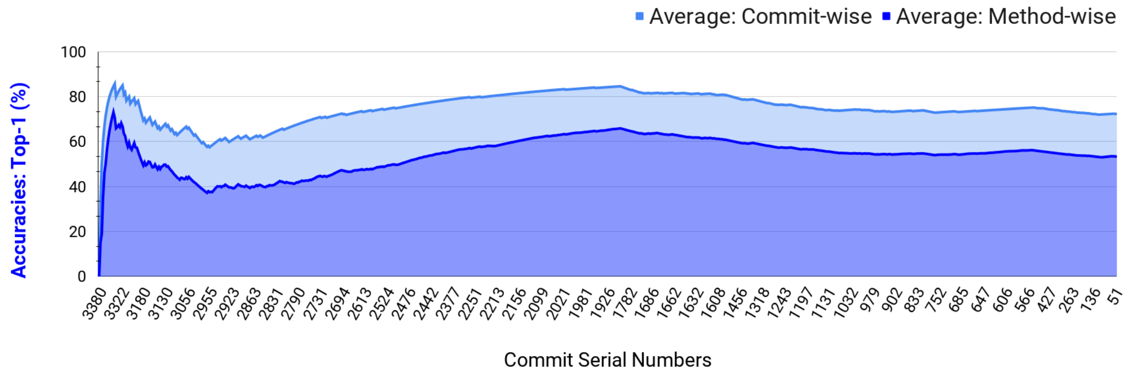
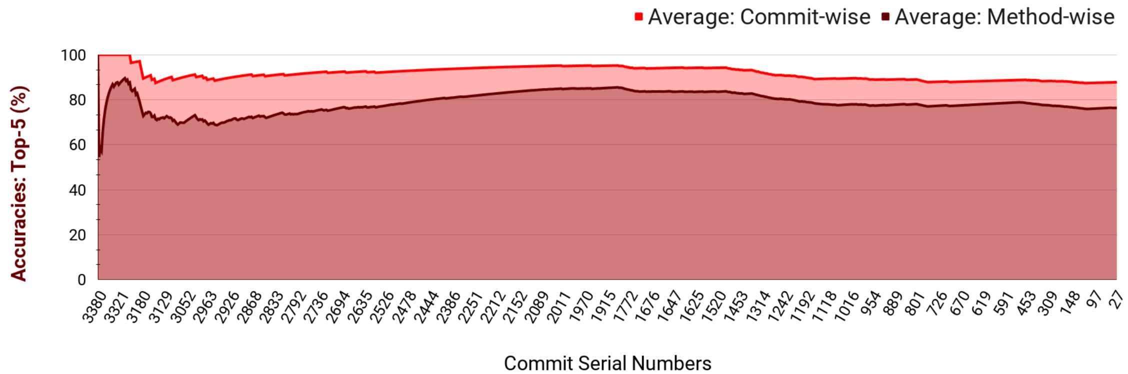


Figure B.7: Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *Guava*

B.2 Apache Commons Collections Graphs

B.2.1 Usage Expertise Graphs





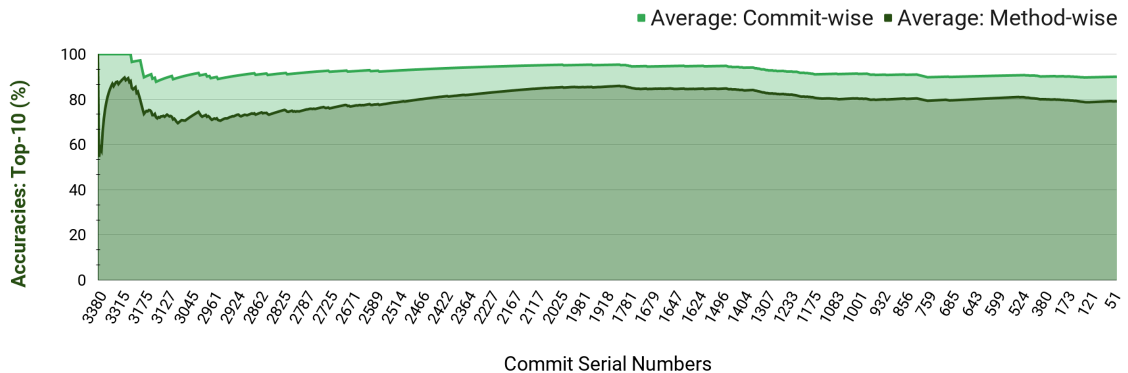
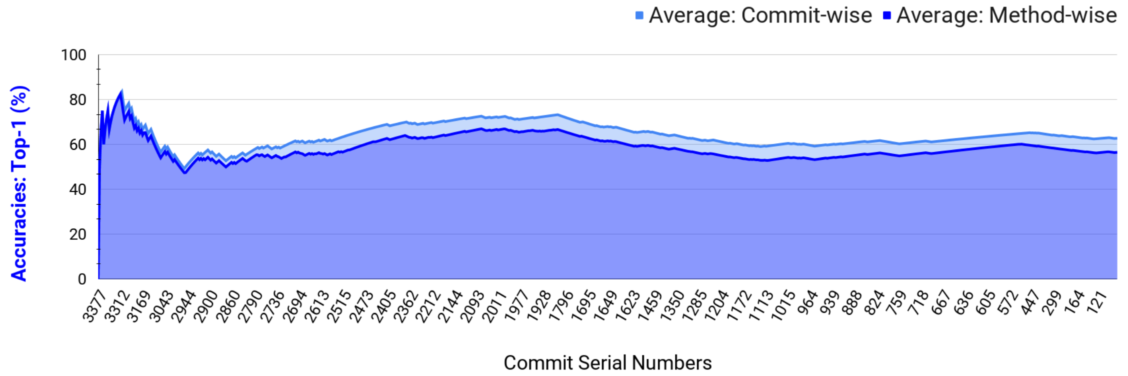
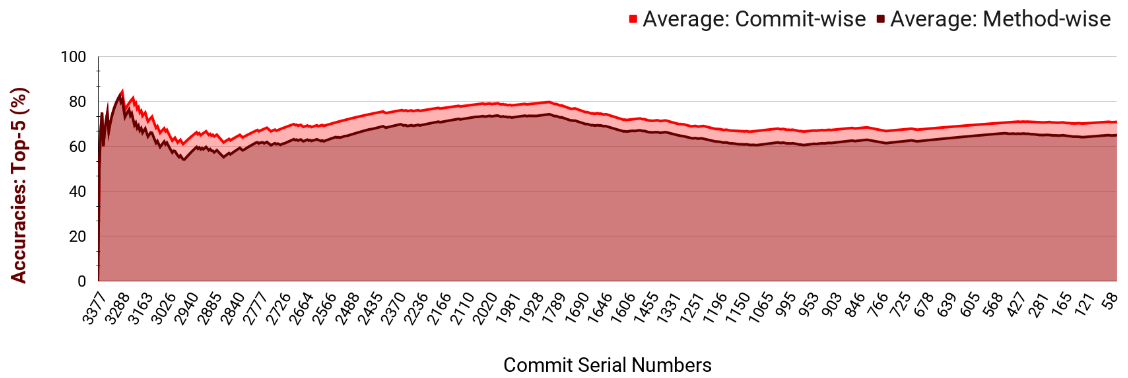


Figure B.8: Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

B.2.2 Implementation Expertise Graphs





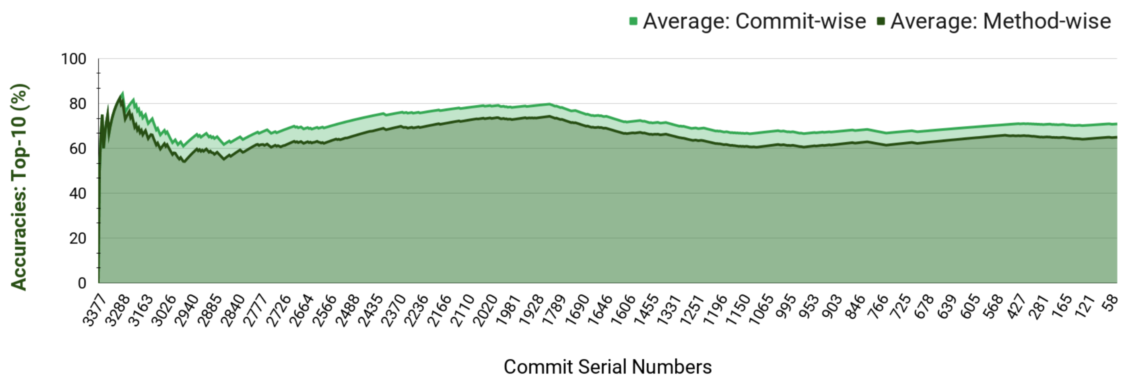
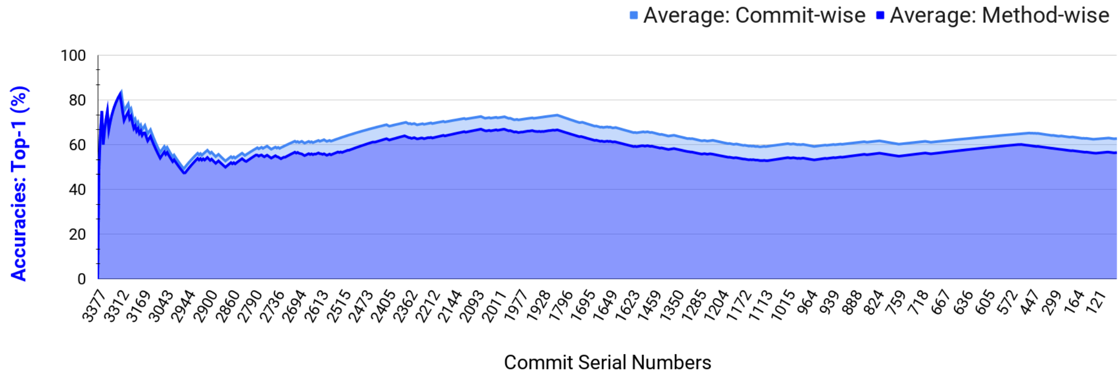
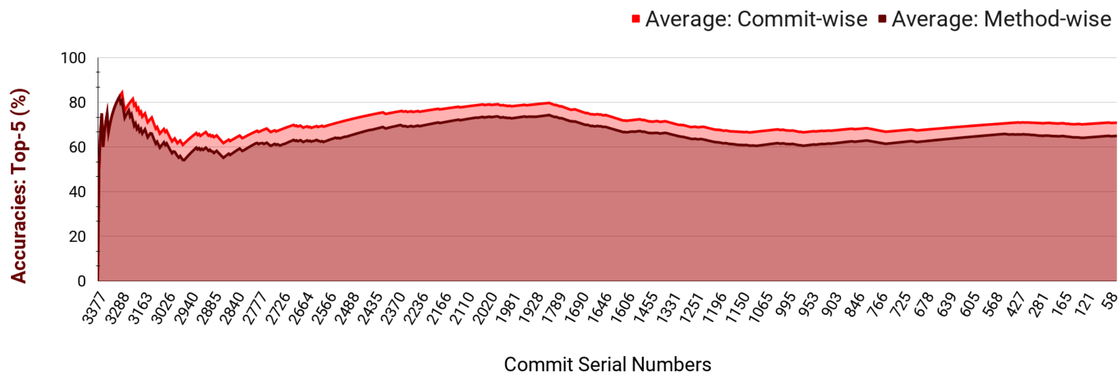


Figure B.9: Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

B.2.3 Combined Expertise Graphs

Weight Set - 1 (w_1 = 0 and w_2 = 1)





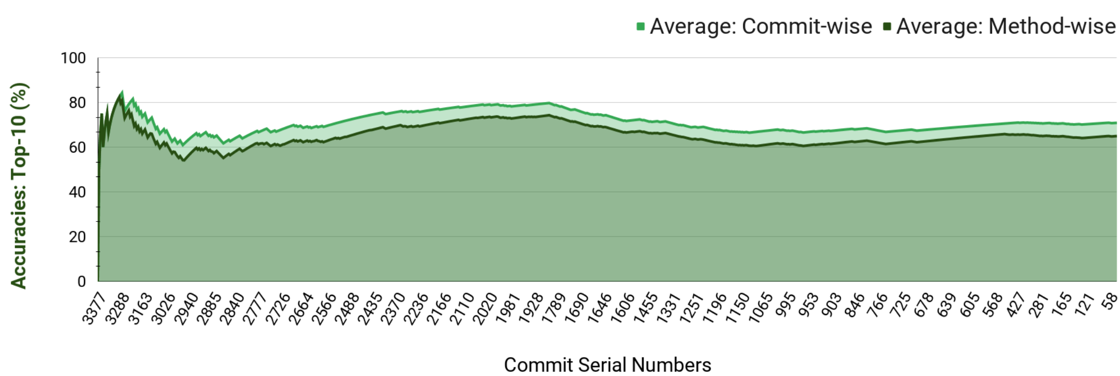
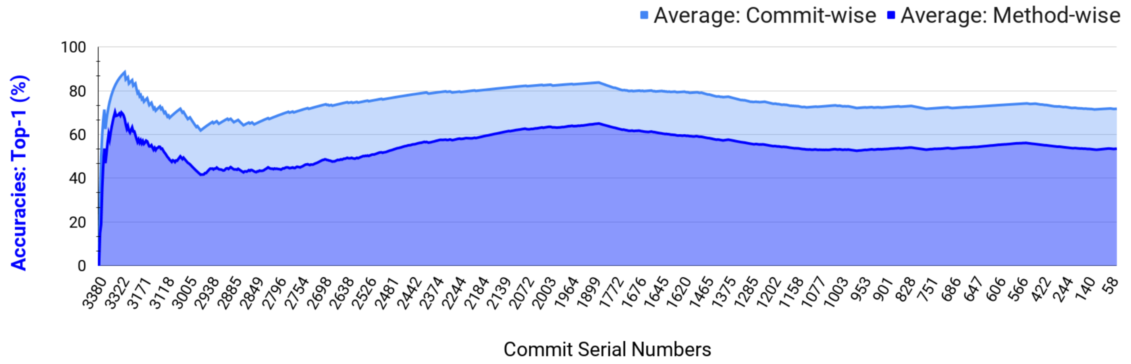
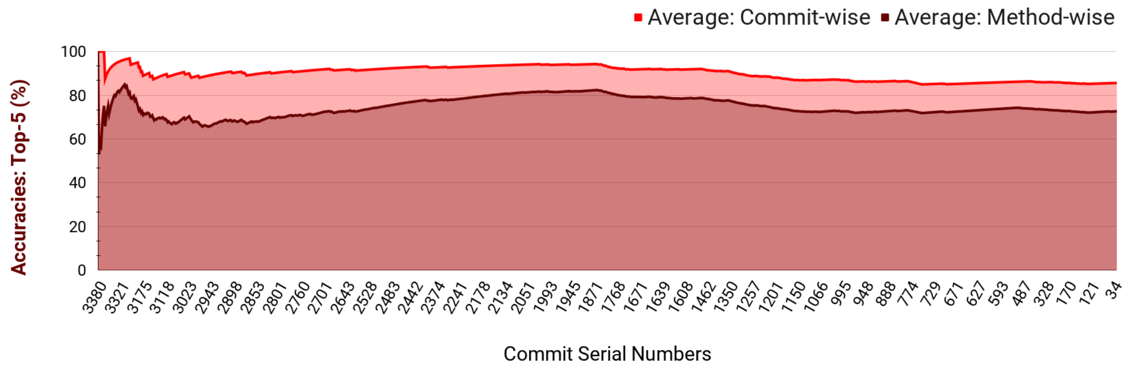


Figure B.10: Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

Weight Set - 2 (w_1 = 0.25 and w_2 = 0.75)





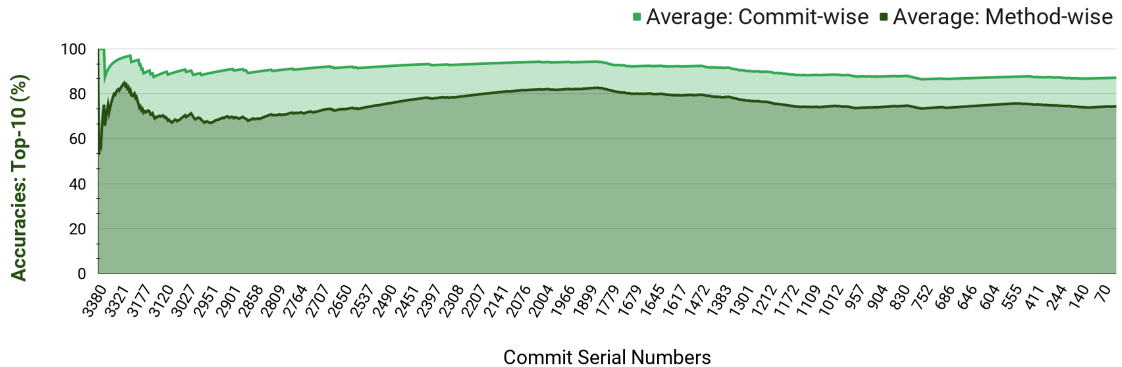
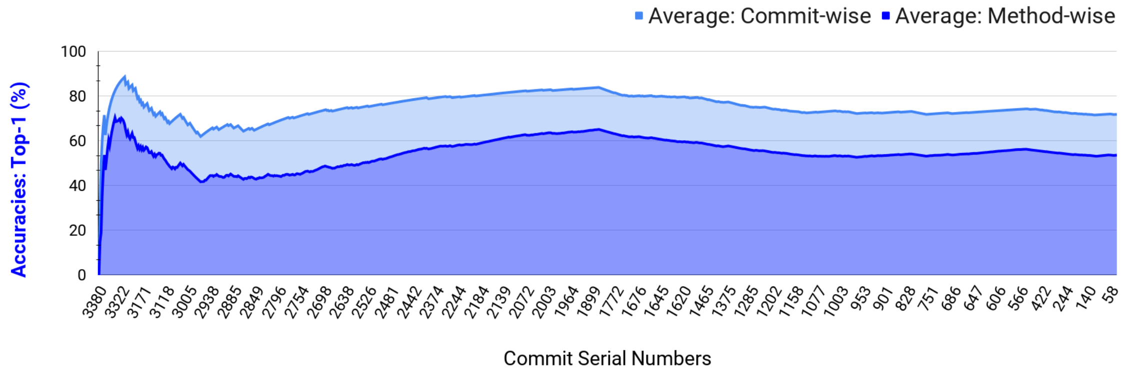
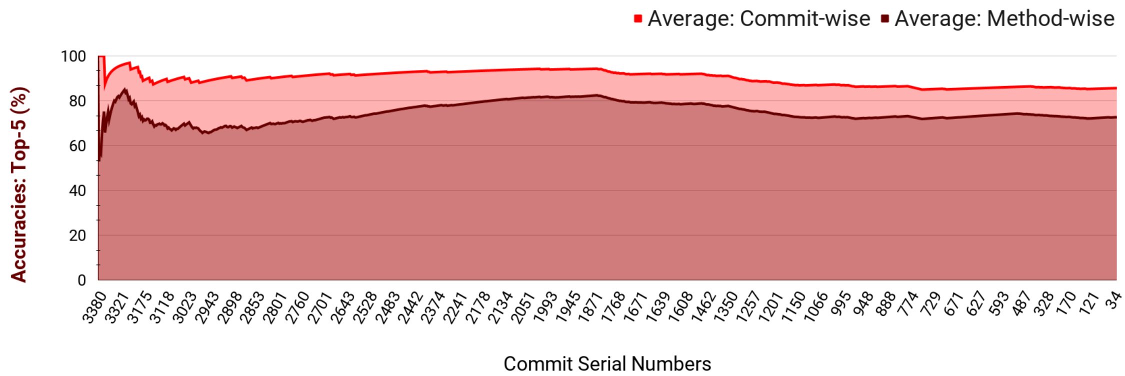


Figure B.11: Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

Weight Set - 3 (w_1 = 0.50 and w_2 = 0.50)





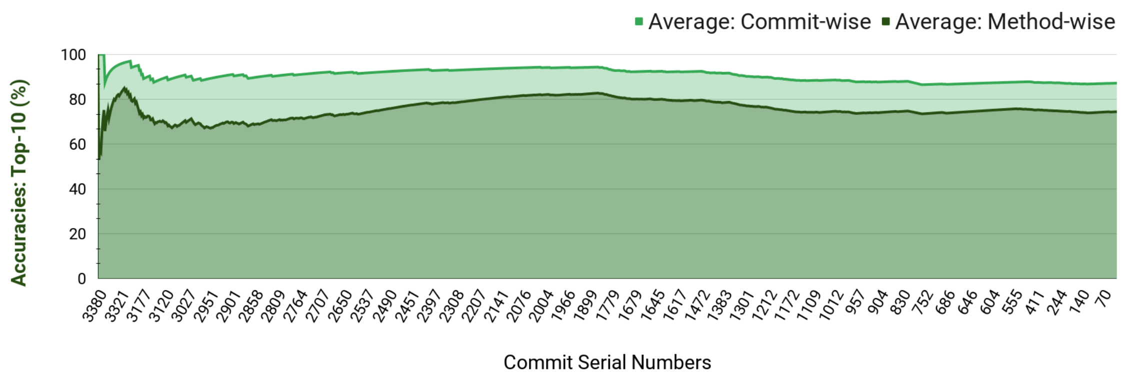
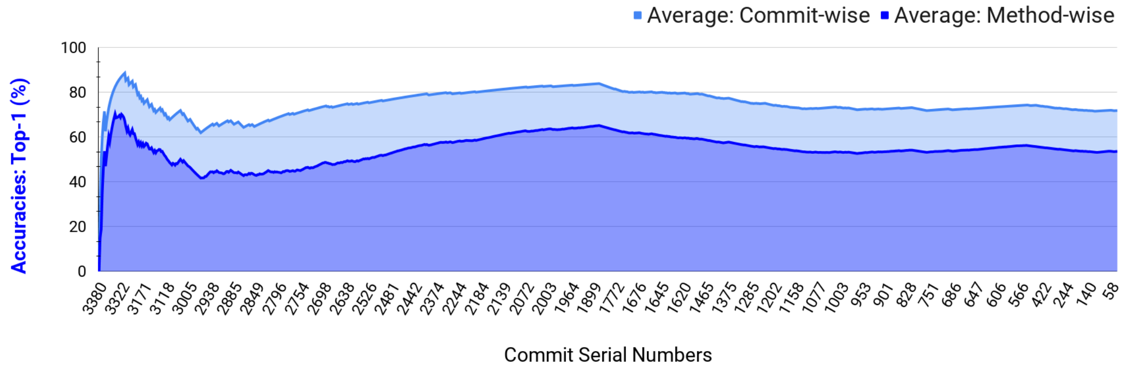
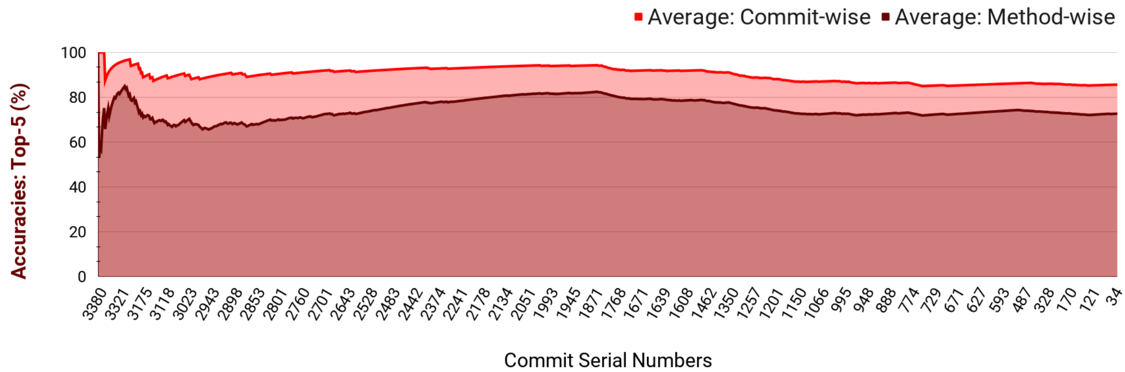


Figure B.12: Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

Weight Set - 4 (w_1 = 0.75 and w_2 = 0.25)





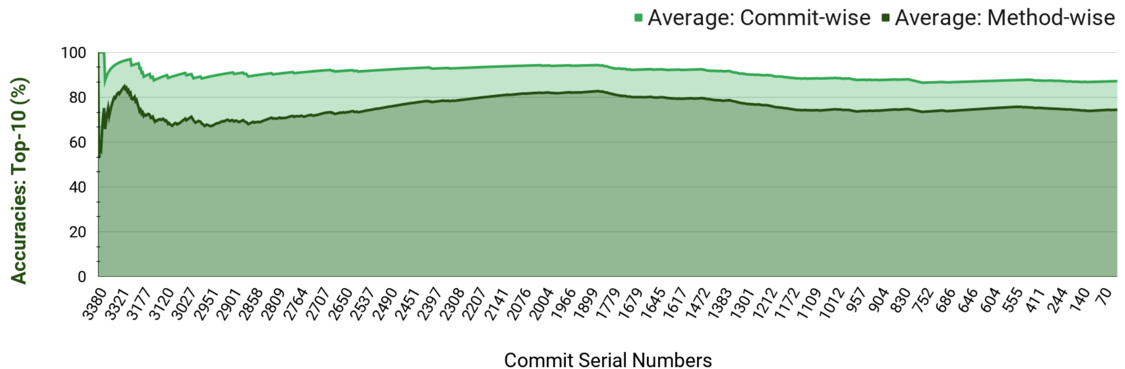
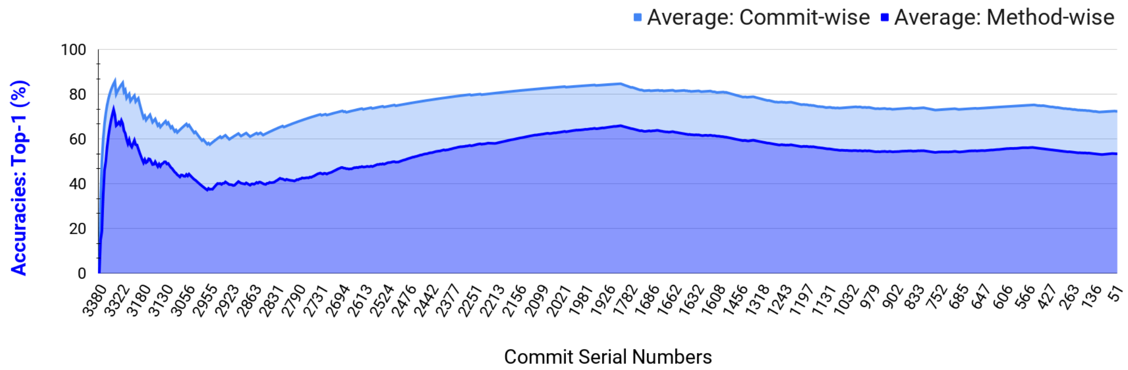
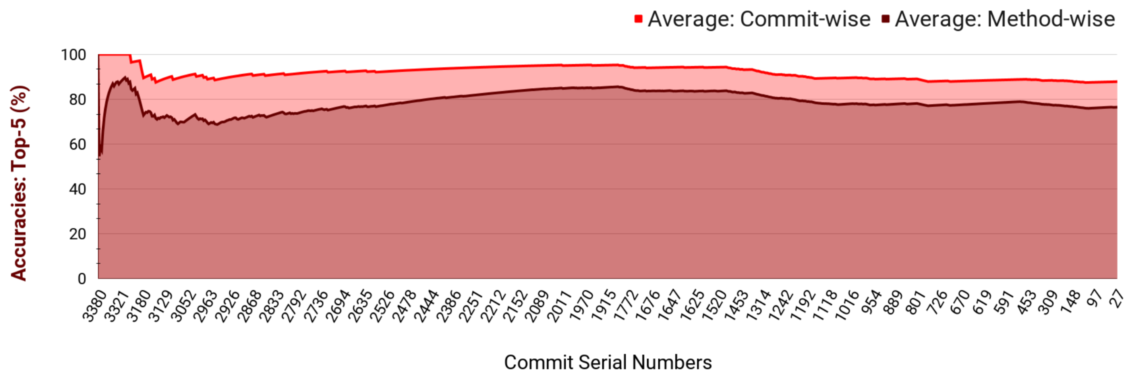


Figure B.13: Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

Weight Set - 5 ($\mathbf{w}_1$ = 1 and $\mathbf{w}_2$ = 0)





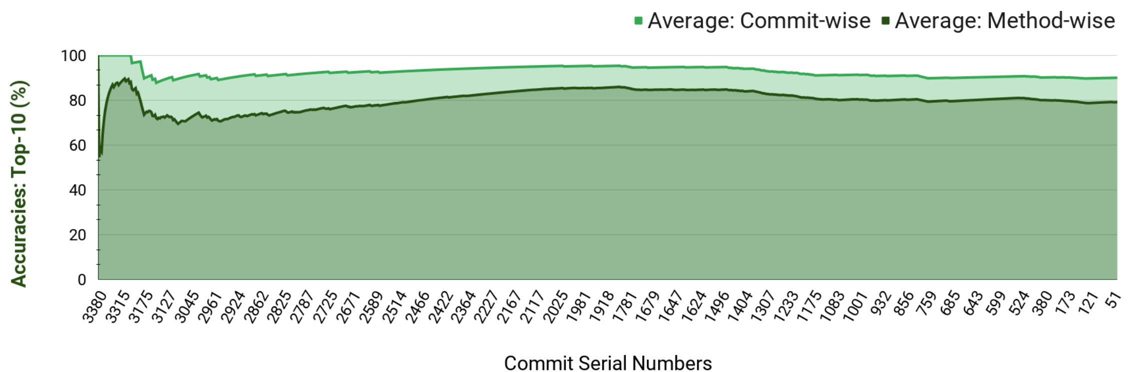
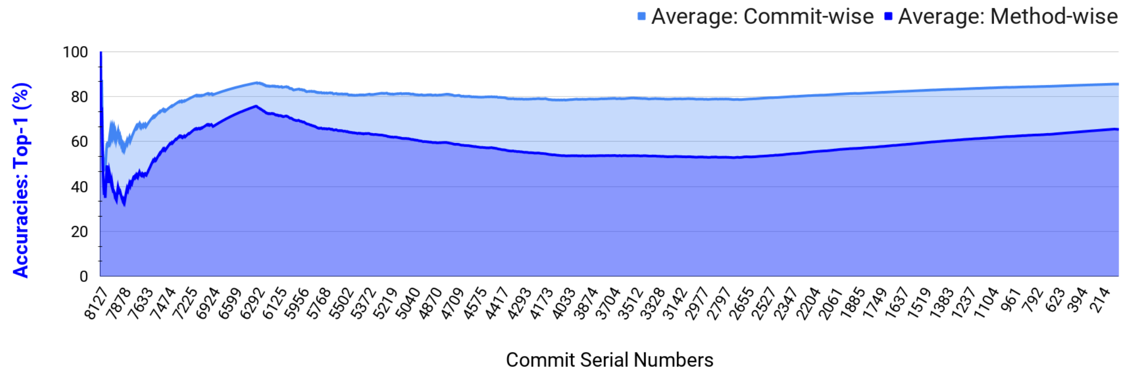
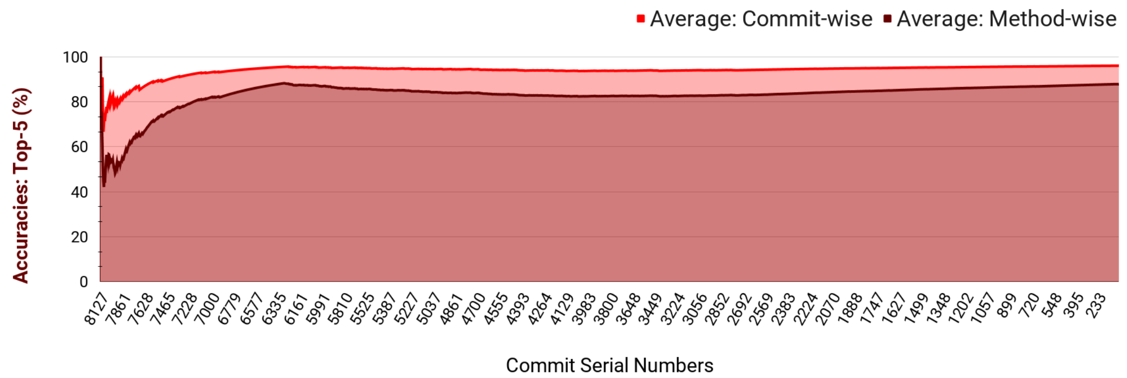


Figure B.14: Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *Commons Collections*

B.3 Eclipse's AspectJ Graphs

B.3.1 Usage Expertise Graphs





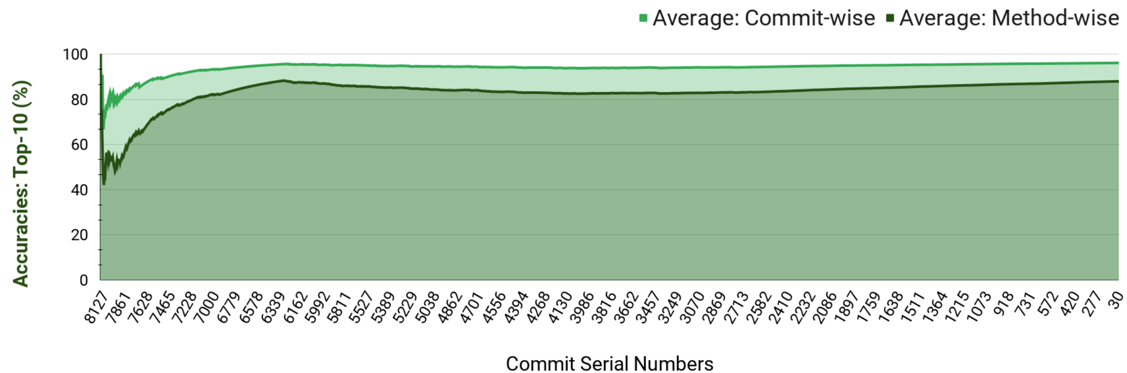
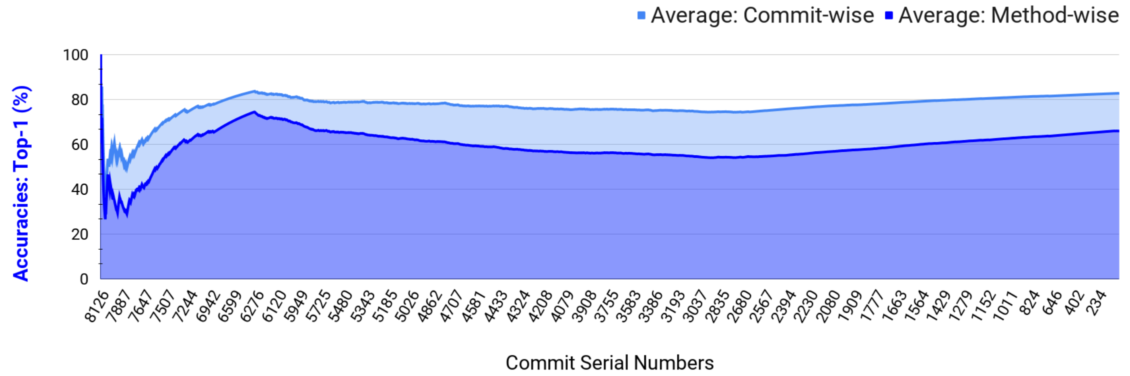
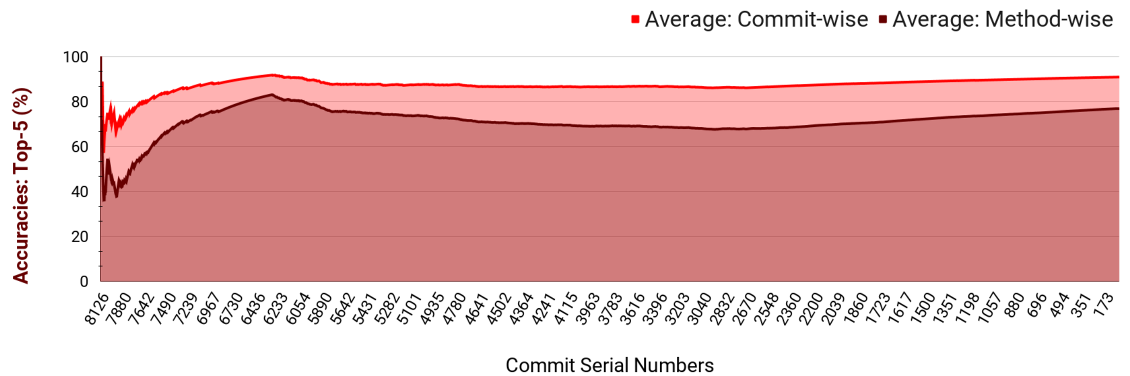


Figure B.15: Average accuracy graphs with Usage Expertise data at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

B.3.2 Implementation Expertise Graphs





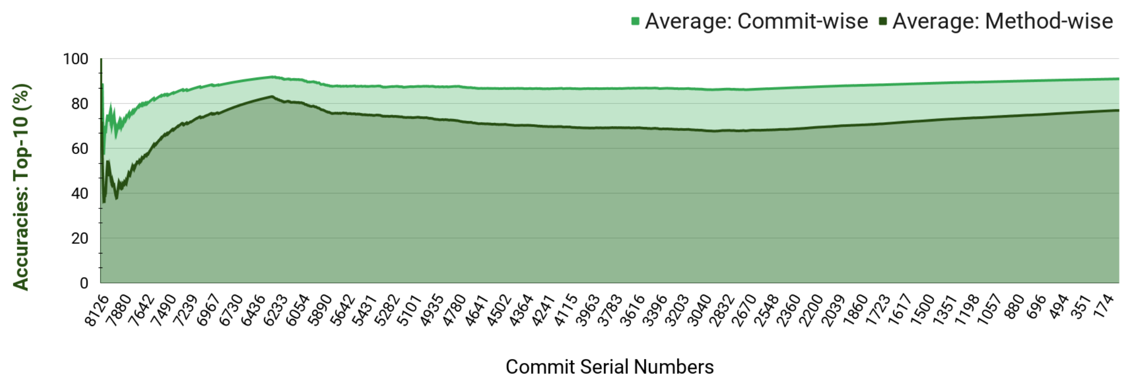
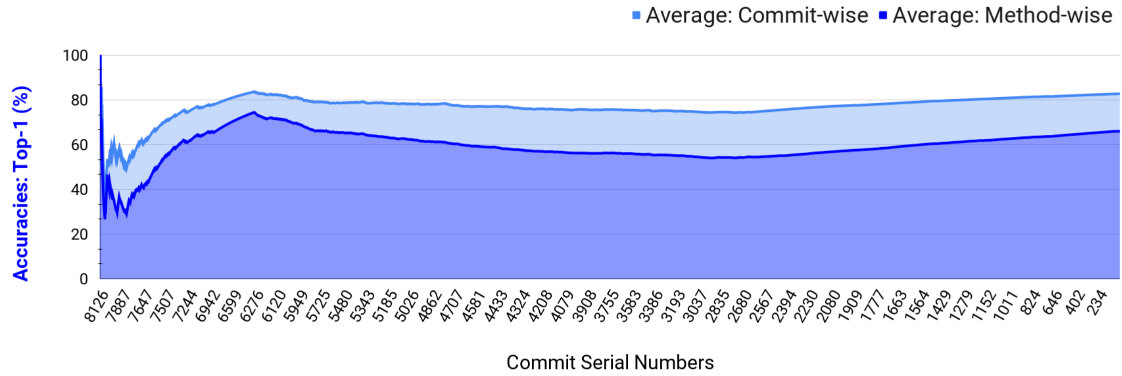
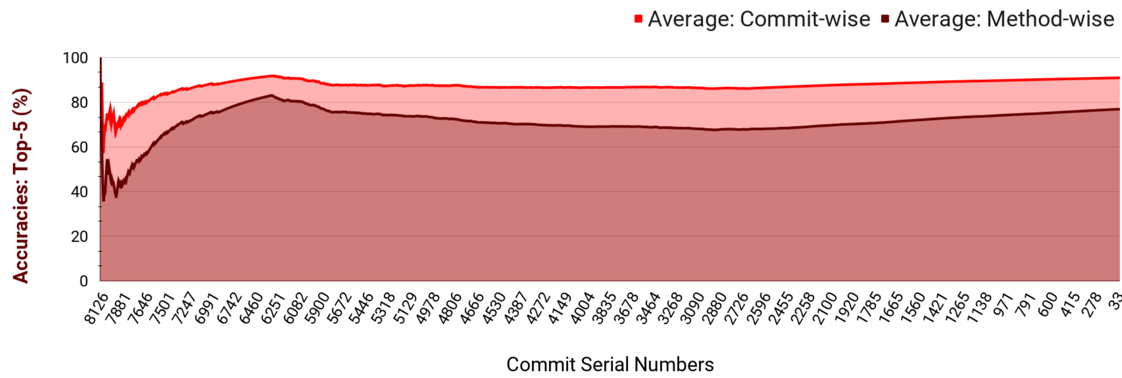


Figure B.16: Average accuracy graphs with Implementation Expertise data at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

B.3.3 Combined Expertise Graphs

Weight Set - 1 (w_1 = 0 and w_2 = 1)





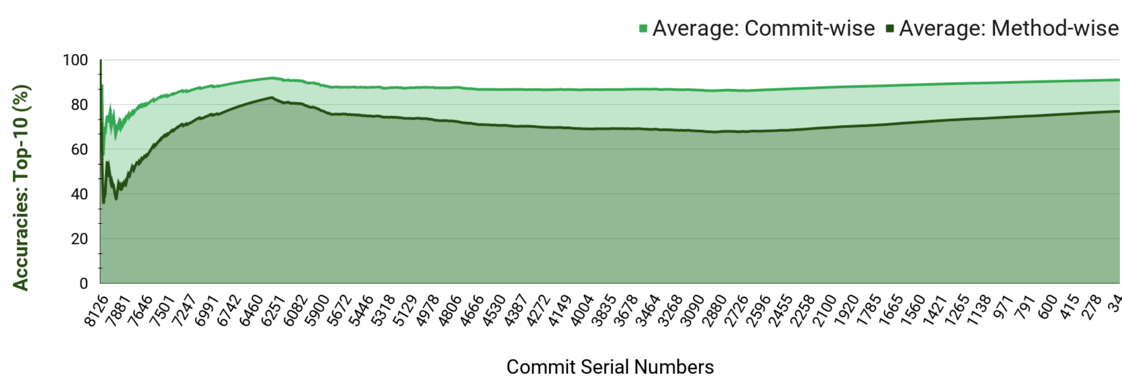
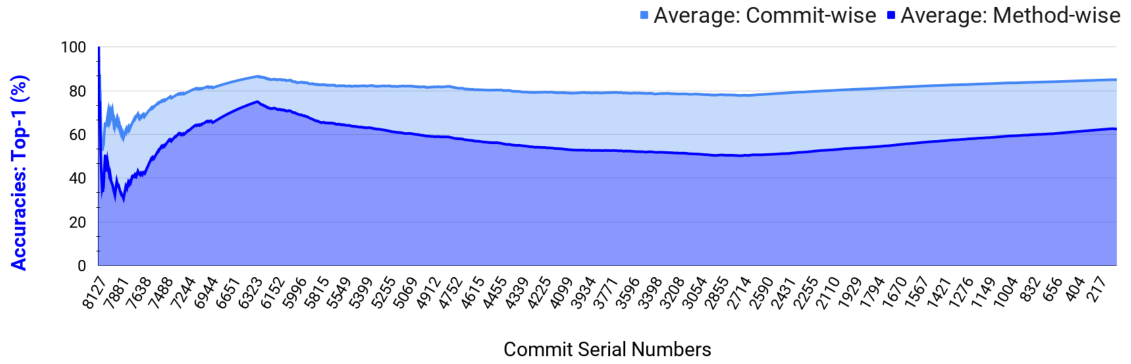


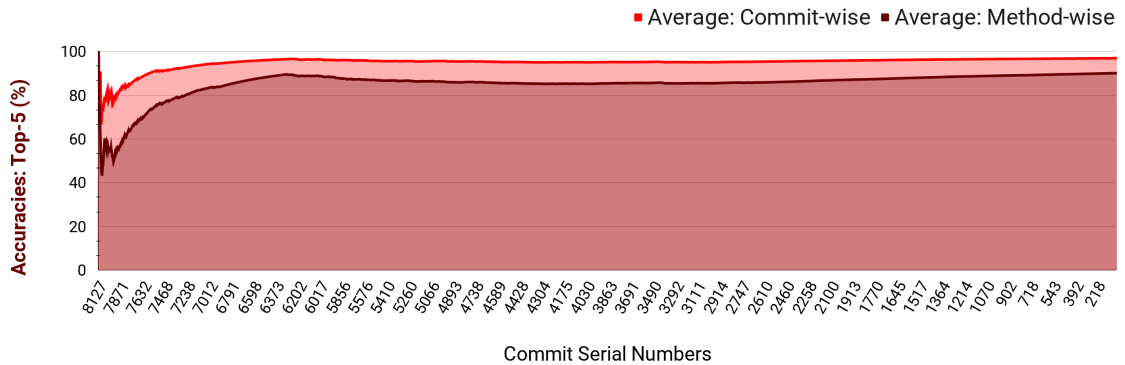
Figure B.17: Average accuracy graphs with Combined Expertise data (WeightSet-1) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

Weight Set - 2 ($w_1 = 0.25$ and $w_2 = 0.75$)

AspectJ: Combined Expertise Data - Weights (0.25, 0.75)



AspectJ: Combined Expertise Data - Weights (0.25, 0.75)



AspectJ: Combined Expertise Data - Weights (0.25, 0.75)

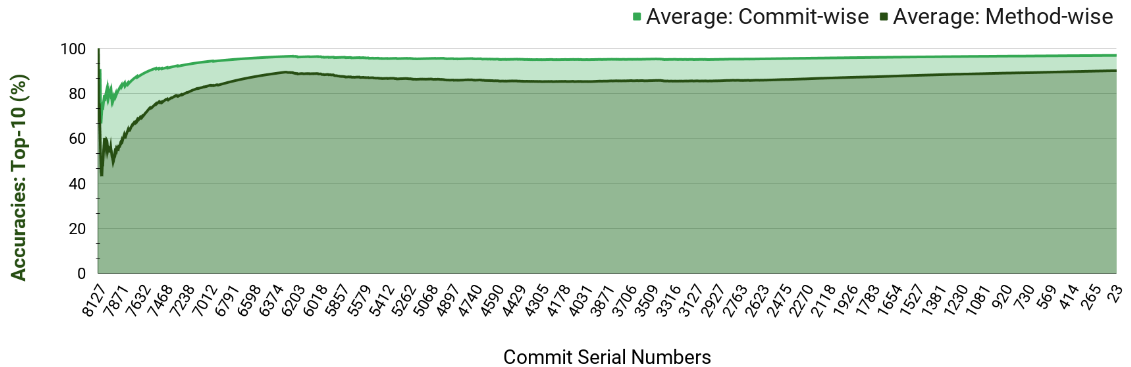
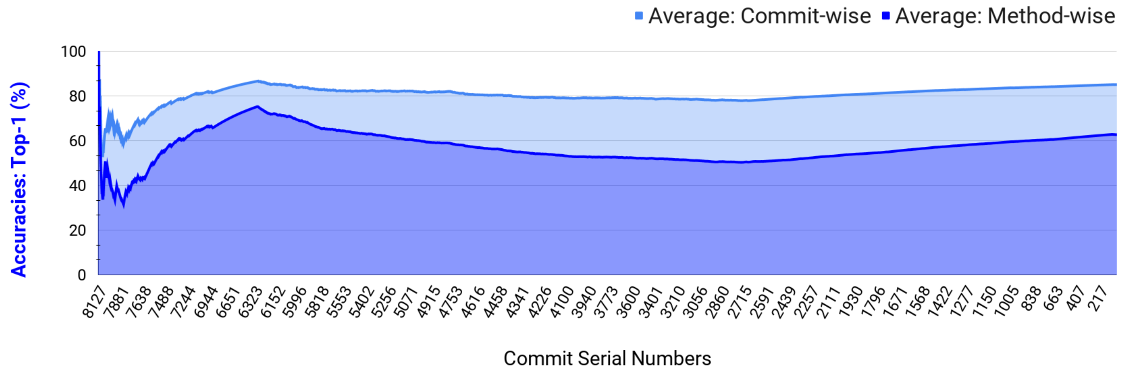


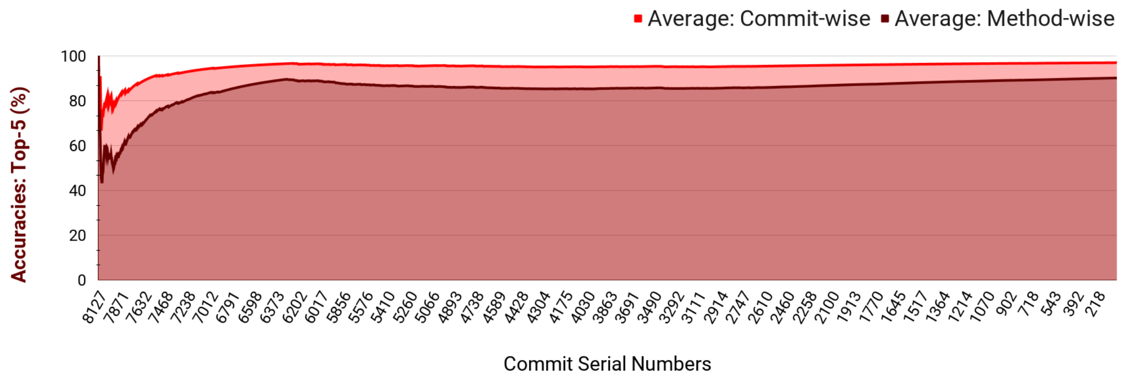
Figure B.18: Average accuracy graphs with Combined Expertise data (WeightSet-2) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

Weight Set - 3 ($w_1 = 0.50$ and $w_2 = 0.50$)

AspectJ: Combined Expertise Data - Weights (0.5, 0.5)



AspectJ: Combined Expertise Data - Weights (0.5, 0.5)



AspectJ: Combined Expertise Data - Weights (0.5, 0.5)

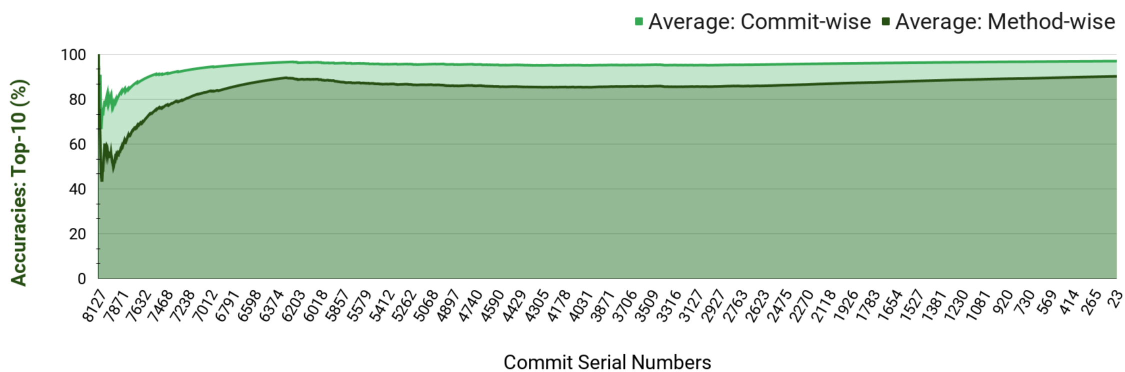
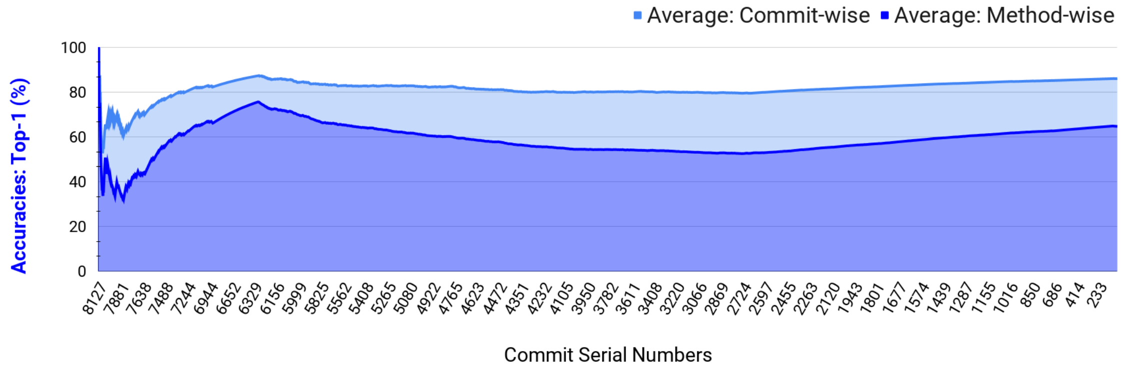
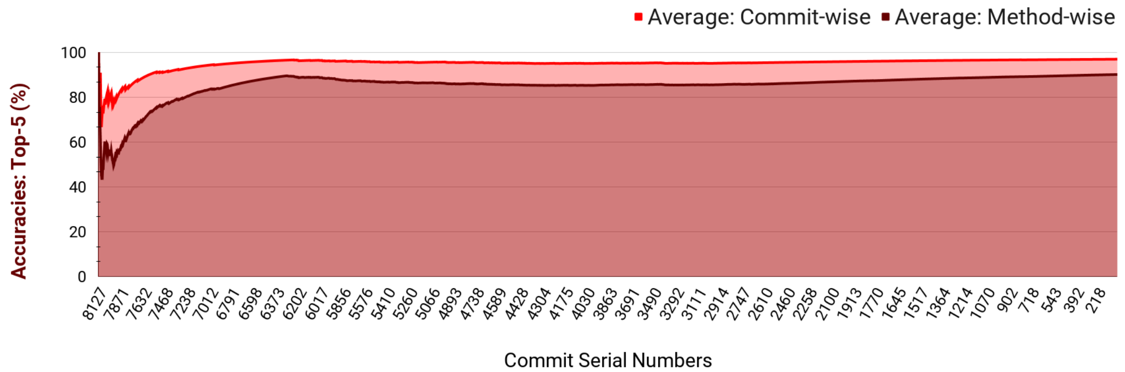


Figure B.19: Average accuracy graphs with Combined Expertise data (WeightSet-3) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

Weight Set - 4 (w_1 = 0.75 and w_2 = 0.25)





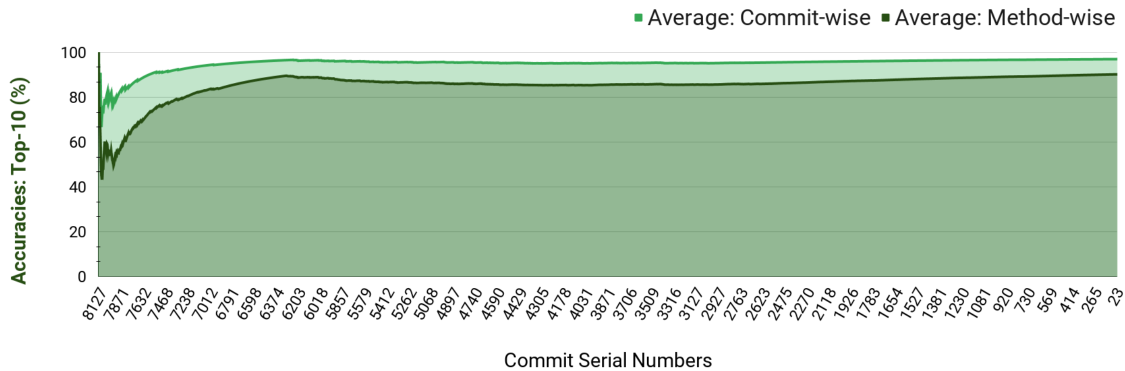
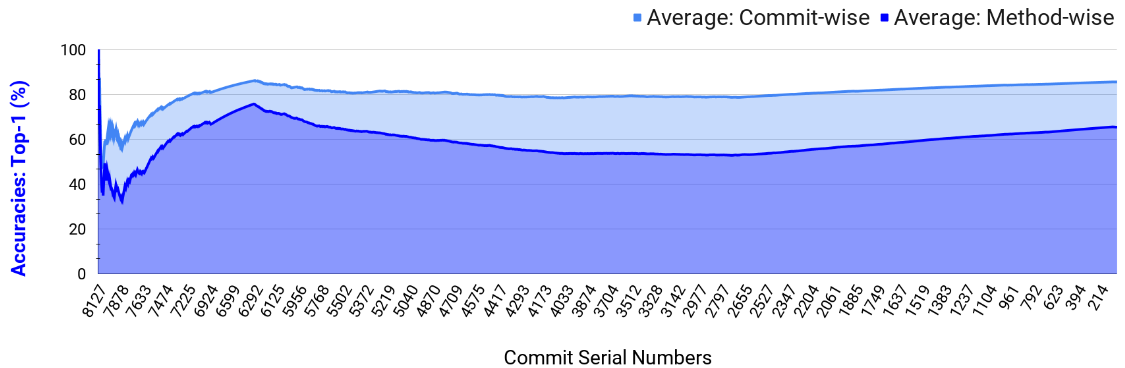
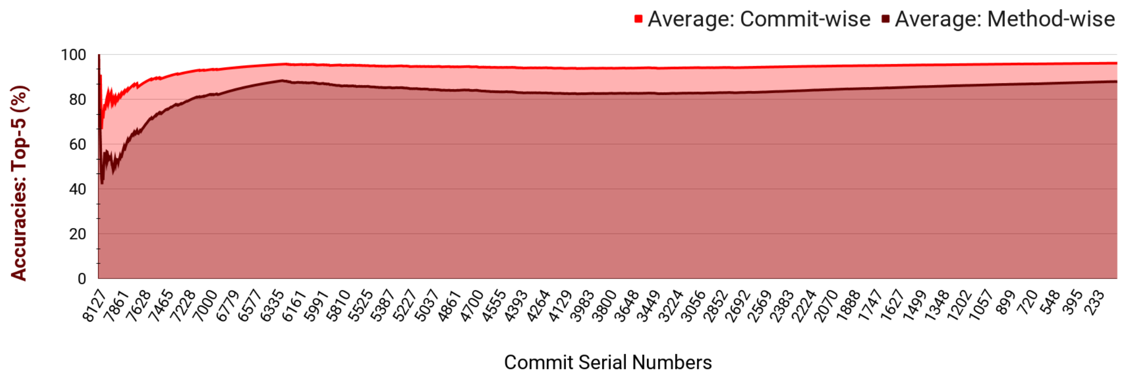


Figure B.20: Average accuracy graphs with Combined Expertise data (WeightSet-4) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*

Weight Set - 5 ($w_1 = 1$ and $w_2 = 0$)





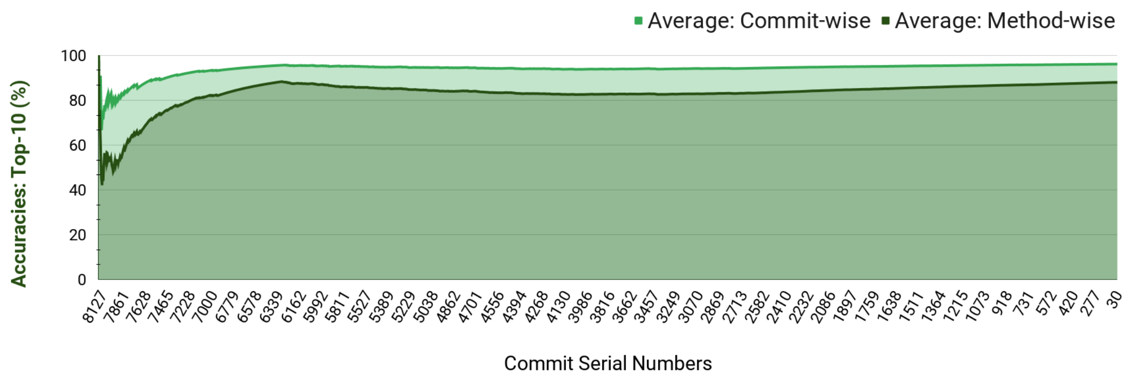


Figure B.21: Average accuracy graphs with Combined Expertise data (WeightSet-5) at Top-1, Top-5, and Top-10 recommendations for project *AspectJ*