

EXPERTS RECOMMENDER SYSTEM
USING TECHNICAL AND SOCIAL HEURISTICS

A Thesis Submitted to the College of
Graduate Studies and Research
In Partial Fulfillment of the Requirements
For the Degree of Master of Science
In the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Ghadeer Kintab

© Copyright Ghadeer Kintab, July, 2013. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

ABSTRACT

Nowadays, successful cooperation and collaboration among developers is crucial to build successful projects in distributed software system development (DSSD). Assigning wrong developers to a specific task not only affects the performance of a component of this task but also affects other components since these projects are composed of dependent components. Another aspect that should be considered when teams are built is the social relationships between the members; disagreements between these members also affect the project team's performance. These two aspects might cause a project's failure or delay. Therefore, they are important to consider when teams are created. In this thesis, we developed an Expert Recommender System Framework (ERSF) that assists developers (Active Developers) to find experts who can help them complete or fix the bugs in the code at hand. The ERSF analyzes the developer technical expertise on similar code fragments to the one they need help on assuming that those who have worked on similar fragments might understand and help the Active Developer; also, it analyzes their social relationships with the Active Developer as well as their social activities within the DSSD. Our work is also concerned with improving the system performance and recommendations by tracking the developer communications through our ERSF in order to keep developer profiles up-to-date. Technical expertise and sociality are measured using a combination of technical and social heuristics. The recommender system was tested using scenarios derived from real software development data, and its recommendations compared favourably to recommendations that humans were asked to make in the same scenarios; also, they were compared to the recommendations of the NaiveBayes and other machine learning algorithms. Our experiment results show that ERSF can recommend experts with good to excellent accuracy.

ACKNOWLEDGEMENTS

First of all, I would like to thank my two talented supervisors Dr. Gordon McCalla and Dr. Chanchal Roy for giving me the chance to combine two interesting majors Artificial Intelligence and Software Engineering. I am grateful for their encouragement, their support, their guidance, and their patience during the completion of this work and thesis.

I would like to thank the internal examiners Dr. Jim Greer and Dr. Julita Vassileva; and the external examiner Dr. Shahedul Khan for giving me their time to set up this defence.

I am also thankful to Manishankar Mondal, Masud Rahman, and Muhammad Asaduzzaman from SRLab for helping me complete my work.

Appreciation is extended to Farouq Al-Omari, Saidur Rahman, and Shamima Yeasmin from SRLab; Edgar Lelei, Mayya Sharipova, and Stephanie Frost from ARIES Lab; and Shaikhah Al-Otaibi from MADMUC Lab for the time they provided to conduct our experiment.

Very special thanks go to my parent and my family for their support, advice, encouragement, patience, and other qualities for which I cannot find the words to describe here.

I would like to thank all my friends in Saskatoon who were my second family during my stay in Canada.

This Thesis is dedicated

To the Greatest Father and the Loveliest Mother

Abdulaziz Kintab and Alia Kurdi

To the Kindest Aunts

Aysha Kintab and Amira Kintab

To the Most Wonderful Siblings

Abdulrahman Kintab, Enas Kintab, Yasmeen Dahlawi,

Anas Kintab, Hussam Kintab, and Atef Kintab

TABLE OF CONTENTS

PERMISSION TO USE.....	I
ABSTRACT	II
ACKNOWLEDGEMENTS	III
TABLE OF CONTENTS	V
LIST OF TABLES	VII
LIST OF FIGURES	VIII
LIST OF ABBREVIATIONS	X
CHAPTER 1	1
INTRODUCTION	1
CHAPTER 2	10
BACKGROUND AND RELATED WORK	10
2.1. RECOMMENDATION SYSTEMS BACKGROUND	10
2.2. RELATED WORK	14
2.2.1. Time-based Technique	15
2.2.2. Modification-based Technique	15
2.2.3. Code Usage-based Technique	17
2.2.4. Dependency-based Technique	18
2.2.5. Similarity-based Technique	19
2.3. LIMITATIONS OF RECOMMENDER SYSTEMS TECHNIQUES AND OUR WORK	20
CHAPTER 3	23
DEVELOPED ARCHITECTURE AND METHODOLOGY	23
3.1. INTRODUCTION	23
3.2. EXPERT RECOMMENDER SYSTEM FRAMEWORK (ERSF)	24
3.3. FROM DESIGN TO IMPLEMENTATION	27
3.3.1. Identifying Experts	27
3.3.2. Experts Recommender System Design	35
3.3.3. Experts Recommender System Implementation	43
CHAPTER 4	57
EXPERIMENT AND EVALUATION	57
4.1. INTRODUCTION	57
4.2. EXPERIMENT OVERVIEW	58
4.3. EXPERIMENT METHODOLOGY	59

4.3.1. Collecting Human Rankings Phase.....	59
4.3.2. Determining Heuristic and Group Weights Phase	72
4.3.3. Designing Recommender System Algorithm Phase	94
4.3.4. Evaluating the Accuracy of the Algorithm Phase	95
4.4. RESULTS	106
4.4.1. Heuristic and Group Weights.....	106
4.4.2. Algorithm Accuracy.....	112
4.4.3. Limitations	120
CHAPTER 5	123
CONCLUSION AND FUTURE WORK.....	123
5.1. SUMMARY	123
5.2. LIMITATIONS AND FUTURE WORK.....	125
5.2.1. Improving the Human Rankings Collection	126
5.2.2. Conducting the Experiment on More Judges	126
5.2.3. Try other Attributes Selection Methods in Weka	126
5.2.4. Build an ERSF Plug-In	127
REFERENCES.....	128
APPENDIX A	133
HUMAN RANKING COLLECTION.....	133

LIST OF TABLES

Table 3.1: Developer Likelihood to be an Expert Example (Formula-2)	53
Table 3.2: Developer Likelihood to be an Expert Example (Formula-3)	54
Table 4.1: Scenario-1 Judge Rankings.....	74
Table 4.2: Technical Heuristics (the Filtered Attribute Evaluation Method Input Example).....	76
Table 4.3: Scenario-2 Judge Rankings.....	78
Table 4.4: Social Heuristics within the DSSD (the Filtered Attribute Evaluation Method Input Example)	81
Table 4.5: Scenario-3 Judge Rankings.....	83
Table 4.6: Social Heuristics within the ERSF (the Filtered Attribute Evaluation Method Input Example).....	89
Table 4.7: Scenario-3 Groups Judge Rankings.....	91
Table 4.8: Heuristic Groups (the Filtered Attribute Evaluation Method Input Example)	93
Table 4.9: Heuristic and Group Weights by Weka Example	95
Table 4.10: Scenario-1 Rankings (RS Algorithm and NaiveBayes Input Example)	98
Table 4.11: Scenario-1 Judge, NaiveBayes, and RS Algorithm Rankings Comparisons	98
Table 4.12: Scenario-2 Rankings (RS Algorithm and NaiveBayes Input Example)	100
Table 4.13: Scenario-2 Judges, NaiveBayes, and RS Algorithm Rankings Comparisons.....	100
Table 4.14: Scenario-3 Rankings (RS Algorithm and NaiveBayes Input Example)	102
Table 4.15: Scenario-3 Judges, NaiveBayes, and RS Algorithm Rankings Comparisons.....	103
Table 4.16: Scenario-3 Groups Rankings (RS Algorithm and NaiveBayes Input Example).....	104
Table 4.17: Scenario-3 Judges, NaiveBayes, and RS Algorithm Groups Rankings Comparisons	105
Table 4.18: Precision and Recall.....	118
Table 4.19: Collecting Human Rankings	121

LIST OF FIGURES

Figure 2.1: Content-based Recommendations	12
Figure 2.2: Collaborative-based Recommendations	13
Figure 2.3: Hybrid-based Recommendations.....	14
Figure 3.1: Experts Recommender System Framework (ERSF)	25
Figure 3.2: Technical and Social Heuristics	26
Figure 3.3: ECF/DocShare "Share Editor With" Menu	37
Figure 3.4: Sharing Permission Pop-up Window.....	38
Figure 3.5: dev1's Editor Opened in dev2's Framework Example	39
Figure 3.6: ERSF Architecture.....	40
Figure 3.7:UML Class Diagram of the ERSF.....	44
Figure 3.8: XML File from SimCad	45
Figure 3.9: Updated XML File with Developers' Information	46
Figure 3.10: RS MySql Database.....	47
Figure 4.1: Four Phases of the Experiment.....	58
Figure 4.2: The Code Fragment the Active Developer Needs Help On	61
Figure 4.3: Type-1 Clone Fragments	62
Figure 4.4: Type-2 Clone Fragments	63
Figure 4.5: Type-3 Clone Fragment.....	64
Figure 4.6: Developer Technical Expertise Summary	65
Figure 4.7: Scenario-1 Developer Rankings	66
Figure 4.8: Developer Social Heuristics within DSSD.....	67
Figure 4.9: Scenario-2 Developer Rankings	68
Figure 4.10: Scenario-3 Description	69
Figure 4.11: Developer Social Heuristics within ERSF.....	70
Figure 4.12: Scenario-3 Developer Rankings	71
Figure 4.13: Technical Heuristics Weights by Weka.....	77
Figure 4.14: Social Heuristics within DSSD Weights by Weka.....	82
Figure 4.15: Social Heuristics within ERSF Weights by Weka.....	90
Figure 4.16: Heuristic Group Weights by Weka.....	93
Figure 4.17: Technical Heuristics Weights Chart.....	107
Figure 4.18: Social Heuristics within DSSD Weights Chart	108
Figure 4.19: Social Heuristics within ERSF Weights Chart.....	110
Figure 4.20: Groups Weights (Before Using the Recommender System) Chart	111

Figure 4.21: Groups Weights (After Using the Recommender System) Chart.....	112
Figure 4.22: Calculating Precision and Recall.....	113

LIST OF ABBREVIATIONS

DSSD Distributed Software System Development

ERSF Expert Recommender System Framework

CHAPTER 1

INTRODUCTION

A software system is a composite of dependent components. This dependency makes the software complicated especially if the system has a large number of such components; a single developer has limited knowledge, and s/he is unable to work on all the components [16]. This means that other developers are important sources of contact if a developer needs more knowledge about the system [33, 21]. In fact, developers need to cooperate and coordinate both in order to manage system dependencies and build a successful software system. Coordination in software development means “integrating or linking together different parts of an organization to accomplish a collective set of tasks” [16].

However, this coordination will not be successful if a developer or a team manager does not have good experience in identifying and selecting helpers or teammates who have good knowledge to accomplish a task at hand. Begel et al. [4] found that 83% of Microsoft software engineers need on occasion to find most relevant engineers who has knowledge on artifacts (feature, API, product, or service), 56% desires to find software engineers who have worked on the target code before, 56% are concerned with finding who owns a specification and has knowledge about; and 53% care about finding the team who own an artifact they or their team depend on. These needs are the most desired features amongst others, which are concerned with finding some information about the artifacts or finding who might be affected by the changes they make on those artifacts.

Some organizations assign this responsibility to team managers, which might work with a small team. However, with a large team it is difficult to identify each developer’s knowledge and

keep such knowledge up-to-date. In short, identifying and allocating an expert is a difficult problem to deal with [25].

To solve this problem, we developed a recommender system to identify and allocate expert helpers more precisely. A recommender system in software engineering is defined as “a software application that provides information items estimated to be valuable for a software engineering task in a given context” [24]. In our case, the provided information is a ranked list of experts who can help an Active Developer, who is looking for a help, in completing a code fragment at hand.

Those experts need to have good expertise and knowledge of code to understand it and provide help; on the other hand, to have only this expertise without the willingness to help and share the knowledge with other developers might cause problems. Therefore, we developed our approach to consider both technical expertise and social relationships. The technical expertise is based on the concept that developers who have worked on similar code might have knowledge about the problem of the code at hand; thus, they might have the best expertise to complete the code the Active Developer is working on. On the other hand, the social relationships are based on the concept that developers who have worked together in the past know each other and might be more comfortable working together in the future.

The technical expertise is concerned with the knowledge of developers on similar code fragments to the one at hand or the code itself. It assumes that the developers who have worked on similar code fragments or on the code are most likely to be able to help complete the current code. However, since the fragments might be changed by more than one developers, we measure their expertise using four technical heuristics: degree of code similarity, number of fragments, number of lines, and most recent modifications.

Social relationships and activities are concerned with communications between the developers within Distributed Software System Development (DSSD). Our approach considers that developers who have cooperated with the Active Developer in the past might be the best experts to help him/her with the current task. When considering social heuristics and communications with others during the development of a software system; that developer might have better sociality and might be more helpful. In our system, we analyze the past communications between the developers with the Active Developer as well as their social activities within the DSSD. Moreover, we track their communications through our Expert Recommender System Framework (ERSF) to analyze who are good helpers and who the developers are interested to ask for help in order to improve the performance and the recommendations of our recommender system as the follow:

First, from the past communications within the DSSD, we try to understand the relationships between the Active Developer and other developers, and also we study how much each of these developers is active within the whole DSSD. We then measure their sociality using four heuristics; two heuristics measure the strength of the developers relationships to the Active Developer (number of shared files, number of shared commits), and the other two measure how active the developers are within the DSSD (number of shared files, and number of shared commits).

Second, from the communication through our ERSF, we can understand the relationships between the developers, and then measure their closeness to the Active Developer and their sociality within the ERSF using eight heuristics. Four heuristics measure the strength of the developers relationships to the Active Developer (trust of the Active Developer in others, response to the Active Developer, developers who have helped the Active Developer, and

recommended developer to the Active Developer), and the others measure how active the developers are in the ERSF (developer trust, developer response, developer helpfulness, and how often developers are recommended).

To measure the developer expertise and sociality using these heuristics, we first need to extract the required data for these measurements. The data are extracted from different sources depending on the specific group of heuristics (technical heuristics, social heuristics within the DSSD, or social heuristics within the ERSF). First, the data for the technical heuristics are extracted using the SimCad Clone Detection Tool [28], which produces an XML file with similar code fragments (clones) to the one the Active Developer needs help on. We then update that XML file to include information about developers who have worked on those clones. This information is extracted from the Git Repository for each line within the fragment, and it includes the developer name and date of the last modifications. The data for the social heuristics within the DSSD are also extracted from the Git Repository. Finally, the data for the social heuristics within the ERSF are extracted from an RS MySQL database maintained by the ERSF.

After that, we apply the heuristics on these extracted data to measure the developer expertise and sociality. Then, we apply our recommender system algorithm on these measurements for each developer to find his/her likelihood to be an expert to help the Active Developer. Finally, we rank those experts according to their likelihoods from the previous step and provide a ranked list of experts to the Active Developer.

To keep our system up-to-date, we embedded our system in the Eclipse Communication Framework/DocShare Plugin (ECF/DocShare)¹, which provides a communication channel between developers and lets them to share their editors. We track the developer activities within the ECF plug-in and our ERSF and save the information in the RS MySQL database. These data

¹ http://wiki.eclipse.org/DocShare_Plugin

are then extracted to design the heuristics of developer communications within the ERSF as we explained above.

A crucial point that we have considered in creating our algorithm is that not all of the previous heuristics have the same priorities and importance between each other within a group (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF) and not all of these groups are estimated equally. Therefore, we conducted an experiment in order to study the importance of these heuristics and their weights. The study was done based on human graduate student judgments. We ran our experiment with 10 students (whom we call them judges in this document) from Artificial Intelligence, Software Engineering, and Social Computing laboratories at the University of Saskatchewan. The judges were given a list of developers with their technical and social characteristics that represent the technical and social heuristics we are concerned with in our work. However, since we have 16 heuristics in total (four technical heuristics, four social heuristics within the DSSD, and eight social heuristics within the ERSF), we divided these heuristics into three scenarios; each scenario is concerned with a specific group of heuristics as follows:

In the first scenario, the judges were given a piece of code and were asked to assume that it is the one they need help on. There are five similar fragments to the original one with different degrees of similarity, the developers' names who have worked on them, and when the last dates of modifications were. To assist the judges to make their decisions, we summarized each developer's expertise on those fragments with some data representing the four technical heuristics (degree of code similarity, number of fragments, number of lines, and the most recent modifications). At the end, the judges were asked to rank the first three experts they thought were the best to contact and get help from; in addition, they were asked to select the reasons,

among the four technical heuristics, of why they thought that those experts were the best to contact.

Scenario-2 is concerned with studying the four social heuristics within the DSSD (number of shared files and commits with the Active Developer as well as number of shared files and commits within the DSSD). Thus, the judges were given a list of developers and some data representing the four heuristics for each of these developers; they were also given the technical expertise from Scenario-1, so the judges could consider them in their decisions. Like Scenario-1, the judges were asked to rank the first three developers they thought were the best to contact and get help from; in addition, they were asked to select the reasons (from the social heuristics or “Has worked on similar code fragments (Scenario-1)” option for the technical expertise) describing their rankings.

The last scenario is Scenario-3, which is concerned with the eight social heuristics within the ERSF (trust, response, help, and recommended with the Active Developer as well as within the ERSF). The judges in this scenario were given a list of developers with some data representing the eight heuristics for each developer; in addition, they were also given the data from Scenario-1 and Scenario-2 to be considered in the decision making. At the end, the judges were asked to rank the first three developers they thought were the best experts to contact with and get help from as well as the reasons (from the social heuristics within the ERSF, “Has worked on similar code fragments (Scenario-1)” for technical heuristics, and “Has good sociality (Scenario-2)” for the social heuristics within the DSSD) of their rankings.

Besides using these human rankings and the considered heuristics in weighting them as to their importance within their groups and the importance of the groups themselves among each other, we also used the human rankings and the considered heuristics to evaluate the accuracy of

the recommender system algorithm. This is done by comparing our algorithm's rankings to the judges' rankings as well as to the predicted rankings by NaiveBayes, NiveNet, and J48 machine learning algorithms [29].

In the first comparison, we compared which ranking (i.e. 1, 2, or 3) our algorithm gave to a developer with particular expertise and/or sociality to the ranking a human gave to that developer for a particular piece of code. Then, we determined the compatibility between the two rankers (the proposed algorithm and the human ranker). Finally, we used these comparisons to calculate the precision and the recall in order to measure the accuracy of the recommender system algorithm.

The second comparison is concerned with evaluating our algorithm using the predicted rankings by NaiveBayes, NaiveNet, and J48 machine learning algorithms. To do that, we applied the algorithms in Weka^[2] to the human judge rankings. Then, instead of comparing our algorithm ranking for a developer with particular expertise and/or sociality to the human ranking as in the first comparison, we compared the algorithm ranking to the predicted ranking by the machine learning algorithms to determine the compatibility between the two algorithms. Finally, we used the results to calculate the precision and the recall in order to define the accuracy of our algorithm in recommending experts.

Our experiment in analyzing the heuristics and their groups themselves brought many interesting insights regarding the importance of the heuristics within their groups and the groups themselves among each other, as follows:

Within the technical heuristics group, which includes the degree of code similarity, the number of fragments, the number of lines, and the date of last modifications, we found that from the human perspective the developers who have worked on more lines have better expertise in

² <http://www.cs.waikato.ac.nz/ml/weka/>

the code. This is followed by the number of fragments and the most recent modifications. Lastly, the degree of code similarity to the one at hand received the lowest priority by the judges.

Regarding the sociality within the DSSD, the developers who have good relationships with that Active Developer were given more priority than the developers who have social activities within the DSSD (ignoring the bases of these relationships in both cases).

As with the sociality within the ERSF, the developers who have relationships with the Active Developer were weighted higher than the developers who have relationships with other developers within the ERSF. Moreover, trust and response within the ERSF were given more importance than other heuristics.

Regarding the importance between the groups, we found that people are willing to get help from the social developers (either within the DSSD or the ERSF) more than the technically active developers. In addition, people prefer to get help from the developers who are willing to help and have helped in the past (sociality within the ERSF) more than the developers they have worked with (sociality within the DSSD).

From both human rankings and machine learning algorithms, our algorithm shows good to excellent precision and recall in both comparisons.

Comparing to the human rankings, our experiment resulted precision and recall with 40% based on Scenario-1 rankings, 63% based on Scenario-2 rankings, 100% based on Scenario-3 rankings, and 70% based on the group rankings.

Moreover, comparing to the NaiveBayes, NaiveNet, and J48 machine learning algorithms, our experiment resulted in precision and recall of 60% based on Scenario-2 rankings, 70% based on Scenario-3 rankings, and 63% based on the group rankings. However, the precision and recall results between the three machine learning algorithms were different based on Scenario-1

rankings: NaiveBayes were 23%, NaiveNet were 33%, and J48 were 30%; our perspective on these differences on Scenario-1 is the effect of the date of the “Most recent modifications” heuristic. All the data in both Scenario-2 and Scenario-3 are either “nominal” (either “0” or “1”) or “numeric”. However, Scenario-1 is the only scenario that includes, besides the other two types, the “date” type, which represents the “Most recent modifications” heuristic. Therefore, we expect that this is what affected the differentiations of the precisions and the recalls between the three machine learning algorithms, unlike Scenario-2 and Scenario-3 that have the same precisions and recalls between the three of these algorithms.

This good to excellent accuracy based on the precision and recall leads us to recommend our system for use in software system development organization to assist their developers in finding experts who can help complete the code at hand.

The thesis document is organized as follows: Chapter 2 shows recommender system background and related work, Chapter 3 explains the method we developed to recommend experts, Chapter 4 describes our experiment, and Chapter 5 concludes with discussion and suggests future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

Before recommender systems were proposed in software engineering, many studies were conducted to mine the expertise of software engineers [14, 25]. Researchers have benefitted from this and have designed recommender systems to assist developers in organizations to find experts who could help them while they are coding.

In this chapter, we first provide some background for recommender systems in software engineering (section 2.1). Then, we introduce related work in expert recommender systems (section 2.2). Finally, we discuss what the limitations are in existing work and how our system assists in addressing these limitations (section 2.3).

2.1. RECOMMENDATION SYSTEMS BACKGROUND

As technologies have been developed and the number of users has increased, the amount of information has grown and caused information overload. Thus, users sometimes experience difficulty finding suitable information or people to fill their needs. To overcome this overload, recommender systems have been proposed to assist those users in finding interesting information or people to connect with. Such systems include Amazon.com for book recommendations [10], MovieLens for movie recommendations [17], and VERSIFI for news recommendations [5].

Recommender systems became a crucial research topic in the mid-1990s when Hill et al. [8], Resnick et al. [22], and Shardanand and Maes [26] proposed collaborative filtering recommender systems [1]. In 2007, the first conference on recommender systems was held, and by 2009, recommender systems, in general, were defined as:

“Applications [that] aim to support users in their decision-making while interacting with large information spaces. They recommend items of interest to users based on preferences they

have expressed, either explicitly or implicitly. The ever-expanding volume and increasing complexity of information [...] has therefore made such systems essential tools for users in a variety of information seeking [...] activities. Recommendation systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance” [24].

One of the areas where recommender systems have become important and where people often need help completing their tasks is software engineering. Robillard et al. [24] in 2010 defined recommender systems for software engineering (RSSE) as “a software application that provides information items estimated to be valuable for a software engineering task in a given context”.

These information items in software engineering include code, API, or human experts. Many recommender systems have been developed in software engineering to recommend these items to developers, such as CodeBroker [31] for reuse-conductive components, Expertise Browser [19] for recommending expertise of code, Strathcona [9] for relevant examples, ParseWeb [27] for Java code recommendations, Suade [23] for recommending interesting program elements to a developer, eRose [35] for guiding developers to the next change, SemDiff [6] for adaptive change recommendations, and Dhruv [3] for recommending artifacts needed to resolve bugs [24].

Recommendations in most areas could be classified into many categories based on types of information sources and methodologies used to determine these recommendations. Researchers have classified recommendations into three main categories: content-based recommendations, collaborative-based recommendations, and hybrid-based recommendations. Since we are concerned with the software engineering area, we explain these categories from an engineering

perspective. The systems extract developer preferences either explicitly or implicitly and mine their history to determine the most interesting items or software artifacts. However, the history is mined differently in each category as explained below.

1. **Content-based Recommendations:** the systems under this category recommend similar artifacts to the ones developers preferred in the past. Figure 2.1 shows the basic concept of the content-based technique. When the Active Developer looks for an artifact(s) of interest, the recommender system extracts the artifacts s/he liked in the past from his/her profile. Then it looks in the Artifact Profiles for similar artifacts to the ones the Active Developer liked in the past. Finally, the system provides those similar artifacts as recommendations to that developer.

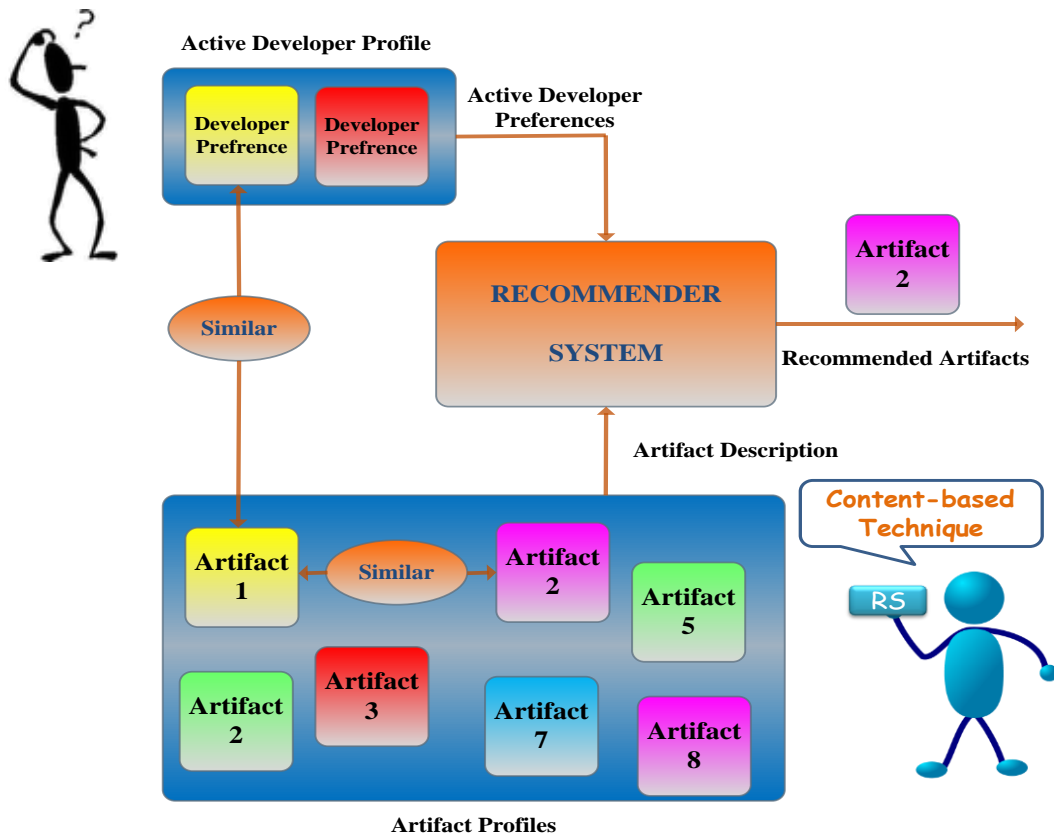


Figure 2.1: Content-based Recommendations

2. **Collaborative-based Recommendations:** collaborative-based recommender systems recommend artifacts that similar developers have preferred in the past. Figure 2.2 illustrates how the recommendations of the software artifacts in this category are designed. When the Active Developer looks for an artifact(s) of interest, the recommender system looks for people who are similar to that developer. This is done by comparing the past preferences of those people to the past preferences of the Active Developer. Then, the system recommends the artifacts in the profiles of the people who have similar past preferences.

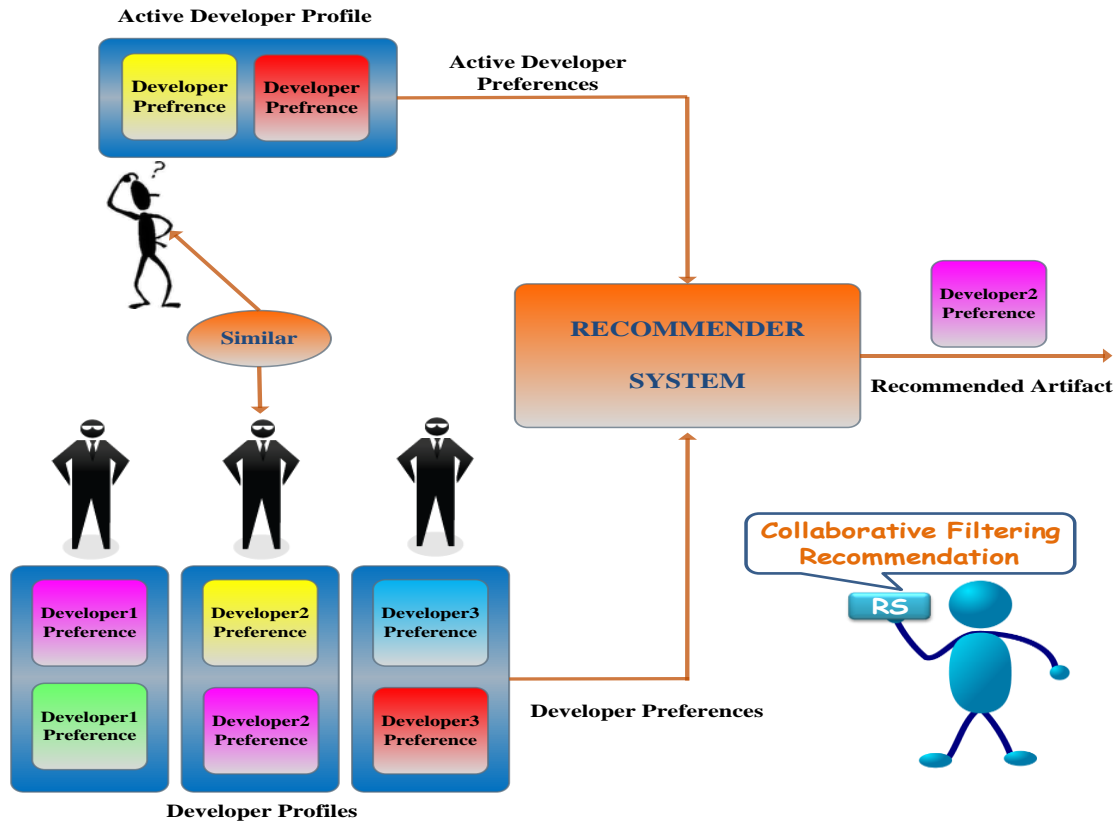


Figure 2.2: Collaborative-based Recommendations

3. **Hybrid-based recommendations:** the recommender systems in this category do not limit their recommendations based solely on similar artifacts (content-based) or on similar people (collaborative-based). Instead, the hybrid-based technique

combines both the content-based and collaborative-based techniques; in other words, it analyzes both the similar artifacts and similar people, as shown in

Figure 2.3.

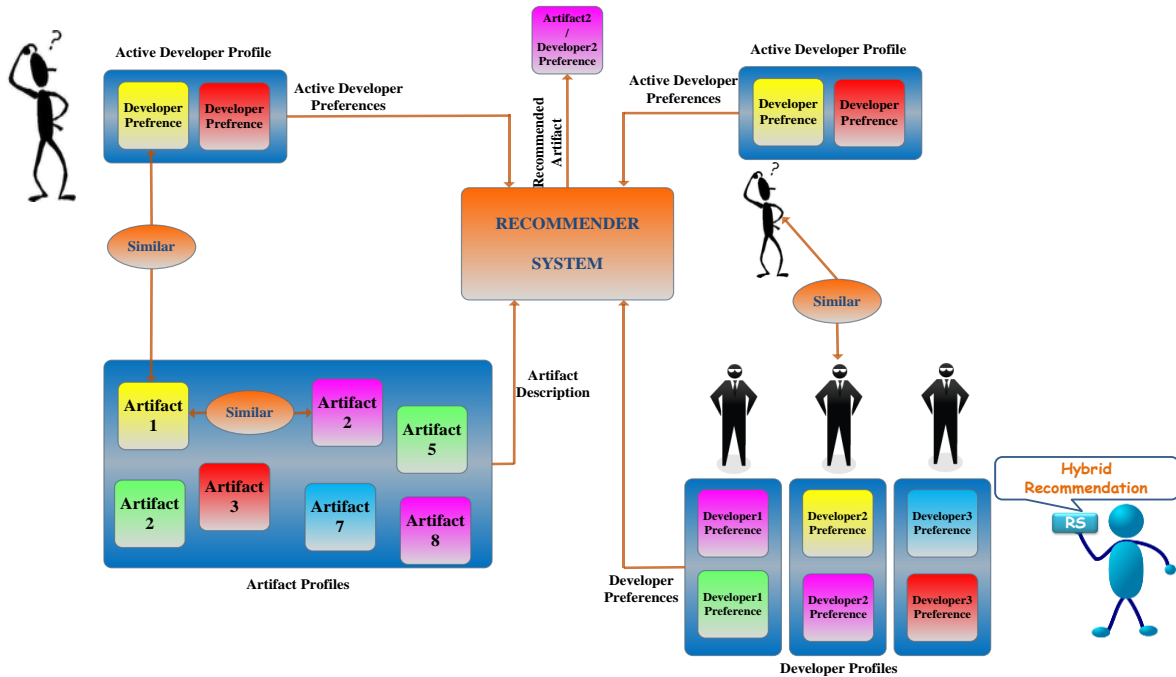


Figure 2.3: Hybrid-based Recommendations

2.2. RELATED WORK

In this thesis, we are concerned with expert recommendation in software engineering. An expert is defined relative to the purpose of a study or a developed system. Alonso et al. [2] describe an expert as “somebody who has contributed a significant number of transactions over time”.

In our work, we define an expert as a developer, other than the Active Developer looking for assistance, who has knowledge on the code at hand and/or on the similar code fragments, has good social collaboration with this Active Developer, or has both the knowledge and the sociality. Expertise in recommender system research is identified using different techniques. We classify these techniques into five categories as follows:

2.2.1. Time-based Technique

In this category, for a given code fragment, developer expertise is identified based on the last time s/he worked on that fragment. If a developer worked most recently on a code fragment compared to other developers, then that developer is considered the expert of that fragment.

From our review, the first recommender system in software engineering was developed in 2000 by McDonald and Ackerman [15,13]. In 1998, they studied developer expertise within an organization [14]. They then used this study to build an Expertise Recommender System in order to suggest developers who had expertise and could help solve the problem at hand. They used the Line10Rule heuristic to determine this expertise. The Line10Rule identifies the person with the most expertise as that person who has modified a piece of code most recently because a developer who has made the most recent changes in code has the freshest code in mind. Thus, this developer is the best one to recommend to help.

2.2.2. Modification-based Technique

Recommender systems under this category analyze added, removed, or edited lines of code to identify expertise. Moreover, some systems benefit from commits on code besides the lines of code themselves to identify expertise, or sometimes they limit the identification based on the checked-in commits on code. Developer expertise is measured based on the size of these modifications or commits.

Mockus and Herbsleb [19] in 2002 built the Expertise Browser Recommender System and used Experience Atom (EA) to identify the expert. Using this system, they track Atomic Change or the simplest change a developer has made on a piece of code or a document. The size of those changes determines a developer expertise.

STeP_IN [32, 33] improved the Expertise Browser by analyzing the social communications between developers as well. The authors built two profiles: Technical Profile and Social Profile. The Technical Profile of a developer is created using Experience Atom (EA) as in the Expertise Browser. However, the Social Profile, which is the new added feature in STeP_IN, is created by tracking the interactions between developers. These interactions are captured by analyzing three components: Inter-Personal Obligation, Total Social Obligation, and Inter-Personal Preference. The first component (Inter-Personal Obligation) captures whether a developer is willing to help somebody who has helped him/her in the past. Total Social Obligation captures if a developer is willing to help the whole group since he has got the most help from others in the past. The last component (Inter-Personal Preference) captures how high a priority a developer puts on collaborating with other developers. From these two profiles, the STeP_IN system identifies and recommends expertise to developers when they need help.

Macek et al. [12] also identify expertise based on Lines of Code (LOC) a developer added, removed, or modified. In addition, they track developer conversation time and duration regarding the code since they consider that developers have knowledge of code they have conversations about.

CodeBook web search [4] uses the modification technique to identify expertise. However, it does not just capture the modification of a piece of code or a document, but it also tracks commits related to this code to identify the expertise. It then uses this information to show the developers who are connected to a searched artifact, as well as other artifacts the “connected” developers have cooperated on.

The check-in commits are also used in the CARES tool [7] to find expertise associated with a file. It shows how many check-ins a particular developer has made relative to others, when the

first and last commits were, if the developer made the most recent check-in, and other information about the standing of this developer.

However, the last two tools (CoodBook and CARES) do not recommend developers but display the developers who have cooperated on a file with their information (number of check-ins a developer has made relative to others, when the first and last commits were, if the developer made the most recent check-in, and other information about the standing of this developer) and allow users to select who they would like to contact.

2.2.3. Code Usage-based Technique

Previous expertise is referred to as Implementation Expertise since it is identified by analyzing developer activities within code or documents, such as lines modified or commits submitted. However, recommender systems under this category do not analyze the implementation of code to identify expertise; instead, they identify this expertise based on their knowledge about calling or using methods.

Ma et al. [11] and Schuler and Zimmermann [25] argue that expertise could be identified not only using Implementation Expertise techniques but also Usage Expertise technique. A Usage Expertise technique assumes that when developers call or use methods, they understand what these methods do with no need to be aware of their implementations; in addition, they have knowledge about the surrounding code these methods are used in. Therefore, the authors build the expertise profile, which includes added or changed method calls and their frequency, based on Usage Expertise. Developer expertise is then measured using four different heuristics: Depth of Method knowledge, Breadth of Method Knowledge, Relative Depth of Method Knowledge, and Relative Breadth of Method Knowledge. Depth of Method Knowledge scores developers according to the sum of calls they have done. Breadth of Method Knowledge assumes developers

who have called a method at least once have expertise on this method; thus, for a set of methods, developers who called the largest number of methods within this set are considered to have the best expertise on these methods. Relative Breadth of Method Knowledge compares the frequency of calls by a developer to the frequency of all other developers. Finally, Relative Breadth of Method Knowledge scores methods that are called with few developers as higher than methods that are called with a large number of developers; thus, developers who called the greatest variety of methods have the best expertise. However, the heuristics are not evaluated to determine which of them provides the best recommendations. The authors found that the Usage Expertise technique produces recommendations with a similar accuracy to that of Implementation Expertise.

2.2.4. Dependency-based Technique

In the dependency technique [18,30], recommender systems identify expertise based on the dependent artifacts (e.g., code, document) to the one a developer needs help on. It considers that developers who have worked on the same or dependent artifacts to the one at hand have knowledge about these artifacts and can help on the current one. Also, it suggests that developers who have cooperated on the same or dependent artifacts should communicate with each other.

Emergent Expertise Locator (EEL) [18] recommends a ranked list of experts based on a set of files that are modified together. The authors analyze congruence between coordination requirements and coordination activities to measure the expertise; i.e., they build their EEL by analyzing how the files have changed in the past (coordination requirements) and who have cooperated on those changes (coordination activities). The system uses a mechanism based on matrices: a File Dependency Matrix, which identifies dependency between files, and a File Authorship Matrix, which represents how many times developers have worked on those files. As

a combination of those matrices, an Expertise Matrix is generated to measure how much expertise a developer has in those dependent files.

The Ensemble Recommendation Tool [30] follows EEL. However, it produces recommendations based on gaps unlike EEL, which produces its recommendations based on congruence. The Ensemble Recommendation Tool uses two heuristics to compute gaps between developers: Arc mirroring suggests that developers who have worked on dependent artifacts should communicate. The Node tie heuristic suggests that developers who have worked on the same artifacts should communicate. These heuristics are then used to compute gaps between developers; the developers with the highest gaps are then recommended to communicate with each other.

2.2.5. Similarity-based Technique

The last technique, the similarity-based technique, finds context similarity between artifacts and identifies experts as those who have cooperated on those similar artifacts and recommends them.

The Conscius [20] recommender system mines a mailing list to recommend experts. Instead of posting a query in a mailing list seeking help, a developer can write a message in the Conscius tool. The tool then takes this message and finds similar messages in the mailing list using a Fuzzy Similarities algorithm. After that, the tool searches for who sent the similar messages and scores them based on how many of these similar messages a developer sent. In addition, developers are scored based on their commits on the classes and dependent classes in these messages.

2.3. LIMITATIONS OF RECOMMENDER SYSTEMS TECHNIQUES AND OUR WORK

Most previous work, such as Codebook [4], CARES [7], Macek et al. [12], Expertise Recommender [15], Emergent Expertise Locator [18], Expertise Browser [19], and Conscius [20] recommend experts based only on their technical expertise in code. However, if developers do not give any importance to the collaboration and are not willing to help, these developers will not be suitable experts to be recommended even if they have high knowledge and expertise about the code. On the other hand, recommending experts based only on social relationships, such as Ensemble [30] will be useless if the recommended developers do not have adequate knowledge of the code even if they have strong relationships with the Active Developer.

In order to address these problems, both technical expertise and social relationships should be considered when designing recommender systems. STeP_IN [32,33] has considered both the technical and social aspects of a recommendation. However, in term of socially, they only looked at the willingness of the helpers to help but not their ability to help. In other words, contacting developers who may not be able to help in achieving the task even if they are willing to help is a waste of time. In addition, STeP_IN only considers the helpers' side, i.e., whether they are willing to help or not, but it does not give any attention to the Active Developers' side, i.e., if they also are willing to contact the helpers for assistance, if they trust them, or if the helpers have good reputations within the organization; in other words, since both parties, helpers and Active Developers, need to communicate and work together, it is crucial to guarantee that they both are comfortable contacting each other and working with other parties and that their communications will not cause any failure during project development.

In order to address the previous limitations, our study developed a hybrid approach, considering both technical expertise and social relationships of the developers, in recommending experts. We designed the technical part based on that fact that “who worked on similar code fragments most probably can understand and help on the code at hand”; we used SimCad Clone Detection Tool to extract those similar code fragments, and we measured the degree of developer expertise using four heuristics: degree of code similarity (clone types), number of fragments, number of lines, and most recent modifications.

Moreover, in our study, we were concerned with improving our recommendations by tracking the developer communications within Expert Recommender System Framework (ERSF), keeping their profiles up-to-date, and using these communications for future recommendations, unlike other methods that limit their expertise and social measurements based on the data in a repository, such as Codebook [4], CARES [7], Macek et al. [12], Expertise Recommender [15], Emergent Expertise Locator [18], Expertise Browser [19], Conscius [20], and Ensemble [30] or in API libraries [11]. In addition, we considered both parties, the Active Developer and the helpers, interest in communicating with each other. We determined helpers’ willingness to help others by capturing their response to the help requests and capturing if they were able to help in the past or not. On the other hand, we determined the Active Developer interest in contacting those helpers by tracking their trust in the helpers. The trust is captured by tracking whom the Active Developer has selected in the past to get help from; we assume that when the Active Developer selects other developers to contact and get help from, s/he trusts them. Moreover, we capture the helper reputations within the ERSF. The last three features (capturing help ability, trust, and reputations) are not considered in the STeP_IN framework, which is the only past work that considers technical and social aspects.

Our method also has other advantages. It can still provide recommendations, even if the code at hand has no authors other than the Active Developer him/herself and has no similar code fragments (clones) by using the social part in the system. Developers can also get help from expert developers, even when completing code in new files, which is not supported by existing studies, such as Codebook [4], CARES [7], Macek et al. [12], Expertise Recommender [15], Emergent Expertise Locator [18], Expertise Browser [19], and Conscius [20], in which they recommend experts for existing files but not for new files. Ma et al. [11] and Schuler and Zimmermann [25] can recommend experts for the new files but by analysing methods in API libraries, unlike our recommender system that analyzing the similar code fragments in a repository. On the other hand, developers with no or little history can also get recommendations since our approach analyzes clones and/or general sociality within the DSSD and the ERSF.

CHAPTER 3

DEVELOPED ARCHITECTURE AND METHODOLOGY

3.1. INTRODUCTION

Software engineers need to coordinate and collaborate successfully in order to develop successful projects in distributed software system development (DSSD). However, since these projects are composed of dependent components, assigning inappropriate developers to work on one of these components does not only affect the performance of this component but also affects other dependent components to this one. Moreover, weak relationships between those software engineers may lead to a project's failure and delay. Therefore, it is important to consider these two aspects when a software engineer team is built.

Some organizations assign this task to team managers or leaders, who might have the ability to find suitable developers to handle tasks at hand within small teams. However, as teams get larger and their structure becomes more complex, finding those developers gets more difficult since team managers or leaders cannot keep track of each developer performance.

Many recommender systems have been proposed in the software engineering area to solve the problem of finding suitable experts to contact. However, some of them just focus on the technical expertise, such as Codebook [4], CARES [7], Expertise Recommender [15], Emergent Expertise Locator [18], Expertise Browser [19], and Conscius [20] or, alternatively, just on social relationships, such as Ensemble [30]. STeP_IN [32,33] is concerned with both technical expertise and social relationships, but it only gives attention to the willingness and not the ability of the helpers to help, as well as it gives attention to the willingness of the helpers to help but not the willingness of the Active Developers to get help from these helpers as we explained in our literature review.

Below are the main contributions of our work in this thesis:

1. We identified experts through:
 - a. Measuring developer technical expertise in code fragments similar to the one that the Active Developer is having problems with or in the code itself if it is authored by other than the Active Developer, and
 - b. Capturing developer social relationships with the Active Developer (who is looking for help) as well as their social activity within the DSSD.
2. We then developed our approach for the Experts Recommender System Framework (ERSF), which combines the technical expertise and social activity to suggest experts to the Active Developer in order to assist in completing the code at hand.
3. We extended our approach for performance improvement purposes. Therefore, we are also concerned with keeping developers' profiles up-to-date in order to provide better recommendations.

The chapter is organized as follows. Section 3.2 gives a brief overview of the ERSF. In section 3.3, we explain in detail our approach starting with identifying expertise and sociality (section 3.3.1), recommender system design and architecture (section 3.3.2), and recommender system implementation (section 3.3.3).

3.2. EXPERT RECOMMENDER SYSTEM FRAMEWORK (ERSF)

This chapter begins with an overview of our ERSF and its main components to help understand its architecture and implementation, which are explained in detail later. The system is composed of four main components as shown in Figure 3.1

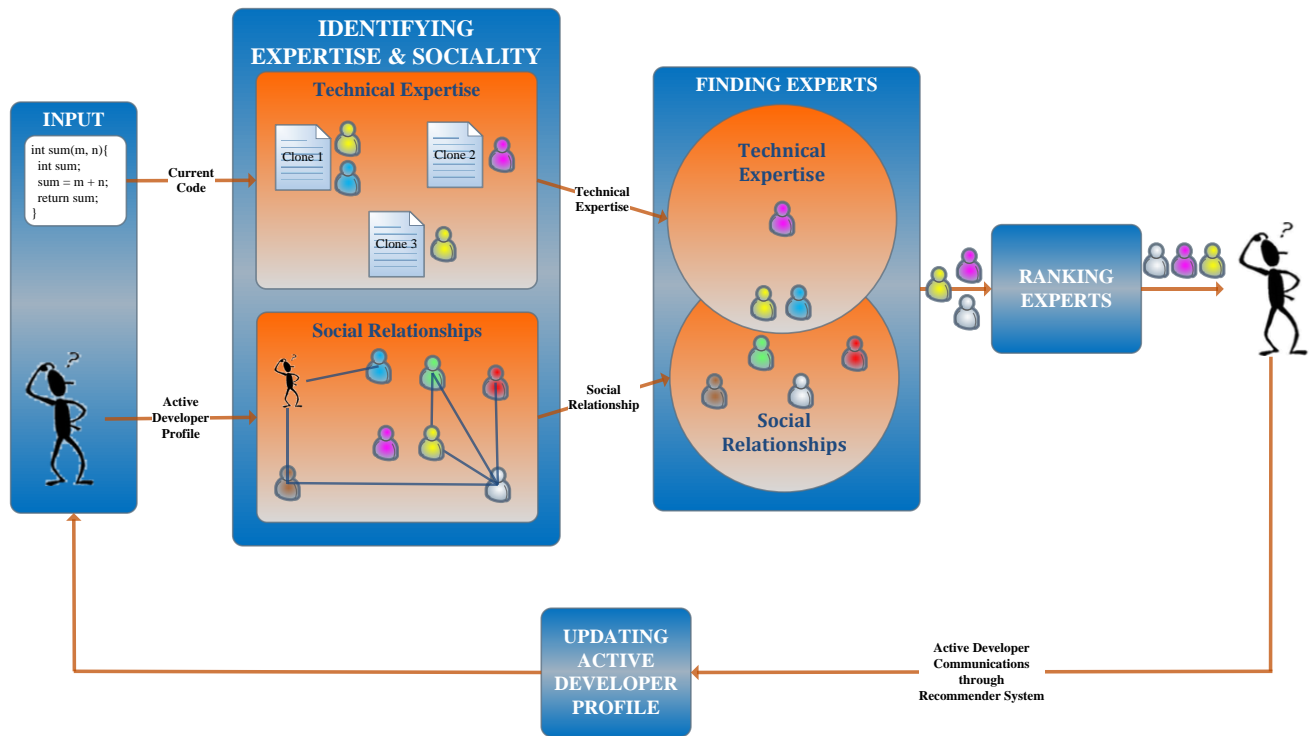


Figure 3.1: Experts Recommender System Framework (ERSF)

Identifying Expertise and Sociality

Our approach to recommend experts is based on heuristics. Technical and social heuristics measurements are used to determine each developer expertise and sociality. The technical heuristics measure developer expertise in code fragments similar to the one being worked on by the Active Developer who needs help. On the other hand, the social heuristics are concerned with measuring developers' relationships with the Active Developer as well as their sociality within the DSSD and the ERSF. The Identifying Expertise and Sociality component measures each heuristic separately for each developer. Figure 3.2 shows the categories of heuristics. Each of these heuristics is explained in detail in the next section.

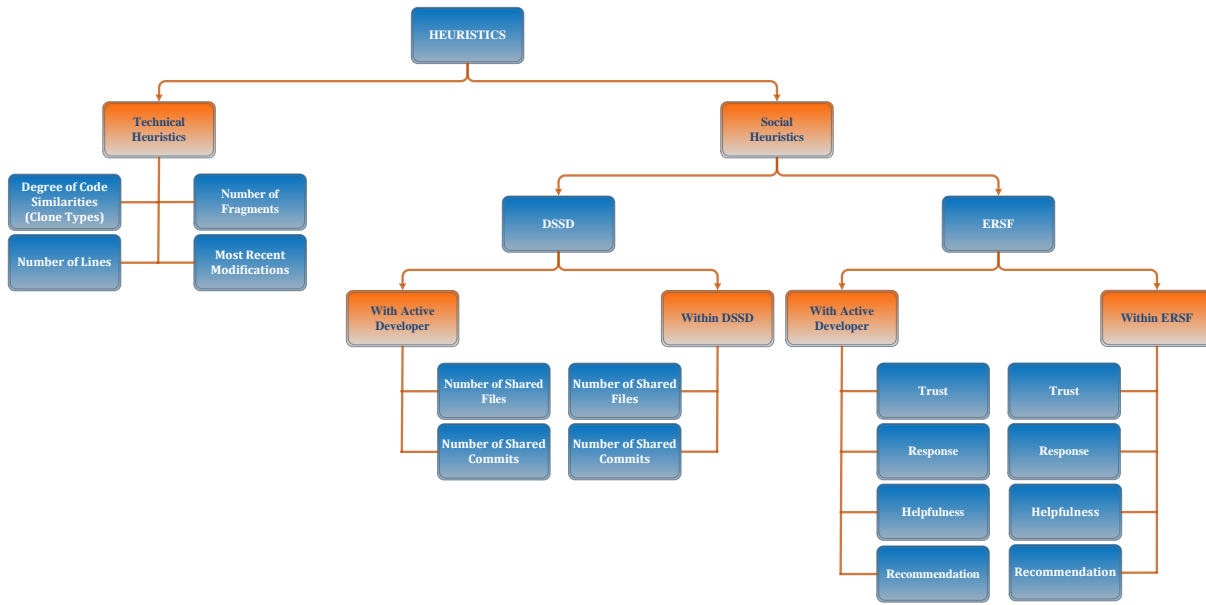


Figure 3.2: Technical and Social Heuristics

Finding Experts

After each heuristic is measured for all the developers, we apply our algorithm that combines and weights these heuristics in order to determine which of these developers should be recommended to the Active Developer to help complete the code at hand.

Ranking Experts

The algorithm from the Finding Experts component might provide a group of suitable experts for the task, and this might cause the Active Developer difficulty in selecting an appropriate expert to contact. For this reason, we add a ranking feature to our approach, so most appropriate developers are ranked first to the Active Developer.

Updating Profiles

One of our contributions, as we mentioned in the introduction, is improving the system performance and recommendations while it is used. We do this by designing the Updating Profiles component. The system in this component tracks the developer communications through our ERSF and adds them to their profiles to keep them up-to-date. Therefore, when developers

use the system, it recommends experts based on the most recently updated profiles, thus providing better recommendations.

3.3. FROM DESIGN TO IMPLEMENTATION

We provided in the previous section an overview of our system framework and its main functionality in order to recommend experts to assist the Active Developer in completing the code at hand. In this section we give a detailed explanation on how each technical and social heuristic is designed to measure each developer expertise and sociality within the DSSD and the ERSF. Then, we explain the system design and architecture, and we conclude with a description of the implementation of the system.

3.3.1. Identifying Experts

We developed our ERSF using the similarity-based technique. The technique combines technical and social heuristics to identify and provide a ranked list of experts who might help the Active Developer in completing the code at hand. The technical heuristics analyze the developer expertise in code fragments similar to the current one, which is extracted using the SimCad Clone Detection tool as we will explain in the design section, or in the code itself if is authored by other than the Active Developer. The social heuristics define the relationships between the developers and the Active Developer as well as the sociality of those developers within the whole DSSD. Another social aspect is concerned with improving the system performance by tracking the developer communications within the ERSF. It uses this information to measure the relationships of the developers to the Active Developer as well as their overall sociality within the ERSF in order to provide better recommendations.

3.3.1.1. Technical Heuristics

We assume in the technical part of the ERSF that developers who have worked on code fragments similar to the one at hand might understand this code better and can help complete it; also, we assume that if this code is written or modified by developers other than the Active Developer him/herself, these developers might be good helpers as well. These similar code fragments are extracted from a repository using the SimCad Clone Detection tool. Then, our system then extracts developers who worked on those fragments and consider them as experts. However, these experts might have different degrees of expertise, which is determined and measured using the following heuristics:

i. Degree of Code Similarity: The code at hand might have one or more similar code fragments in the system. These fragments do not always have the same degree of similarity to the current one. They might be identical with differentiation in white-space or comments, identical with different identifiers' names, or have more or fewer lines compared with the code at hand [35]. We analyze who have worked on each type of those fragments and can help the Active Developers to complete their code.

ii. Number of Fragments: Some of the developers who have worked on similar fragments might have worked on one fragment or more than one. We consider in this heuristic that developers who have modified more of these fragments might have better technical knowledge and can better understand and help with the current code. Thus, developers are considered to have better expertise as the number of fragments they have worked on increases.

iii. Number of Lines: The developers might gain more knowledge about a piece of code as the number of lines they have created or modified increases. We designed this heuristic based on

this concept, so developers are considered to have better knowledge and expertise as the number of lines they have created or modified increases.

iv. Most Recent Modifications: McDonald and Ackerman [15] identify expertise as belonging to the person who has modified a piece of code most recently. They assume that developers who have modified the code most recently are the ones with the freshest code in mind. Thus, these developers might be the best experts to help. We include this heuristic as one measurement of the developer expertise in our work as well.

3.3.1.2. Social Heuristics

The social heuristics analyze the relationships of the developers to the Active Developer and their sociality within the DSSD and the ERSF. These relationships and sociality are extracted from the past communications between the developers and are kept up-to-date through tracking their communications within our ERSF. When the ERSF is used for the first time, the experts are identified using their past communications through the development repository, such as Github³. Then, as the developers use our system to find experts, the system tracks their communications, updates their profiles (stored in an RS MySQL database), and uses them later to improve its performance for recommending experts. Thus, the social heuristics are classified into two categories based on their purpose: social heuristics from the past communications (Git Repository) as a starting point when the system is first used and social heuristics from the communications within the ERSF (RS MySQL database) to improve the system performance and recommendations.

A. Social Heuristics within the DSSD

A repository has a great deal of valuable information about the developers we can benefit from. In our approach we analyze the developer activities in Git repository and construct the

³ <https://github.com/>

social relationships between each other and their social activities within the DSSD. These relationships and activities are then used to recommend suitable experts to the Active Developer to help him/her. However, the Active Developer might have relationships with more than one developers, and more than one developers might be socially active within the DSSD. Therefore, we design the following heuristics to measure the degree of relationships with the Active Developer as well as the degree of their social activity within the DSSD.

a) Heuristics of Developer Relationships with the Active Developer

We design these heuristics assuming that developers might be more interested to ask for help from people they have worked with in the past. On the other hand, the heuristics also assume that developers might be more willing to help people they already know and have worked with.

Therefore, the system uses the constructed social relationships to measure the developer closeness to the Active Developer using the following heuristics:

i. Number of Shared Files: Our methodology in this heuristic builds the relationships using the system files. It considers that if two developers have worked on the same file, these developers might have worked together before, so the system links them in that relationship. Developers are measured to be closer and have stronger relationships with the Active Developer as the number of the files they have shared increases.

ii. Number of Shared Commits: A commit in the Git repository has an author and a committer, who might be the same developer. However, for some commits, the author and committer are two different developers. We consider that since the code changes in these commits have been made with authors who are different from the committers or the commit submitters, the authors and the committers have worked together and thus have social relationships. This heuristic is concerned with the commits the Active Developer has shared

with others, either as an author or committer. The developers become closer to the Active Developer as the number of commits they have shared with the Active Developer increases.

b) Heuristics of Developers Sociality within DSSD

Developers who are active within the DSSD might be ready and more willing to help the Active Developer, even if they do not have any relationships with him/her, or they might have better knowledge on the system and can benefit the Active Developer more than others.

Moreover, this feature also assists developers who have little or no relationships with others to find someone who can help them. We identify and measure developer sociality using two heuristics:

i. Number of Shared Files: This heuristic considers that developers who have cooperated with others on the creation or modification of the system files are social developers. Therefore, for each developer, the heuristic counts the number of files this developer has shared with others. A developer with the largest number of shared files is considered the most social one within the DSSD.

ii. Number of Shared Commits: As we mentioned early, some commits are shared by different developers as an author and a committer. We use this feature as well to measure the developer sociality within the DSSD. The developer gain more sociality in this heuristic as the number of commits they have shared with others increases.

A. Social Heuristics within the ERSF

Improving the system performance is a crucial issue in software system development. In our approach, we are interested in improving our recommendations to the developers as well. In our work, this is done by tracking the developer communications when they use our ERSF and keeping their profiles up-to-date. Moreover, we also design other social heuristics to improve

performance and apply them to the developer profiles to measure their sociality. Below we provide the details:

a) Heuristics of Developer Relationships with the Active Developer

We design these heuristics based on the assumption we explained in the previous section, which is concerned with the relationships of the Active Developer to other developers within the DSSD, i.e., developers who have worked together in the past might be more interested in seeking assistance and helping each other in the future. However, in this section, we measure these relationships using different social heuristics. The data used in these heuristics are extracted from the tracked communications within the ERSF, itself.

i. Trust of the Active Developer in Others: Trust is identified as “a developer who the Active

Developer selects to get help from and who is trusted by this Active Developer”. This

heuristic assumes that one of the reasons the Active Developer might contact another

developer asking for help is the trust the Active Developer has in the other developer.

Moreover, it assumes that a developer whom the Active Developer has trusted most in the

past might be the one whom is going to be trusted in the future. Thus, in this heuristic, a

developer is measured as a closer developer to the Active Developer as the number of times

the Active Developer has trusted him/her increases.

ii. Response to the Active Developer: Responsiveness is identifies as “a developer who

responds to the Active Developer help request is a responsive developer to the Active

Developer”. Developers do not just concentrate to find developers who have good knowledge

about programming or they have worked with before, but also they are concerned with who

responds to their requests. This heuristic is designed based on this concept; it assumes that

developers who have responded to the Active Developer in the past might be ready to

respond to his/her current request. Thus, a developer might be a better recommendation to the Active Developer the more s/he has responded to his/her past requests. However, responding by the developer to the Active Developer in the past does not mean s/he could help him/her; therefore, we also consider the developer's ability to help using the following heuristic.

iii. Developers who have Helped the Active Developer: The main purpose of finding experts is to find someone who can help complete the code at hand. Therefore, it is important to track who were good helpers for the Active Developer. Thus, since these helpers were able to help the Active Developer in the past, they might be the experts who can help the Active Developer with the current task. The developer who has helped the Active Developer the largest number of times is the one closest to him/her.

iv. Recommended Developer to the Active Developer: Begel et al. [4] found that most developers ask their colleagues to recommend others who might help if they do not know the answers. Therefore, the system also provides the recommended developers the ability to recommend others if they are not able to help the Active Developer on his/her request. We benefit from this feature to improve the system recommendations by detecting whom the developers would recommend to the Active Developer and how often they are recommended. The system uses this heuristic as one measurement of the developer closeness to the Active Developer. Developers get closer to the Active Developer the more they have been recommended to him/her by other developers within the ERSF.

a) Heuristics of Developer Sociality within the ERSF

In the previous section we mentioned the importance of measuring the developer sociality within the DSSD. Also, we mentioned that in our methodology we are concerned with improving our system performance and recommendations. Therefore, in this section, we explain which

heuristics we use in order to measure the developer sociality and improve the system recommendation within the ERSF.

i. Developer Trust: Developers trust other developers because they might have good knowledge or might be good helpers within the ERSF, which is what developers need when they are looking for someone to help them. Therefore, in this heuristic we measure the developer sociality as how much they have been trusted by others. Developers with the highest trust might be the most social ones and the best ones to recommend.

ii. Developer Response: If developers do not respond to others' requests when they need help, even if they have a great deal of knowledge in programming, these developers might not respond to the current request, so they should not be recommended. Therefore, in this heuristic, we measure how much the developers are ready and willing to help others by measuring how often they have responded to requests for help in the past.

iii. Developer Helpfulness: When developers use our system to find experts, they look for someone who can help complete the code at hand. Thus, this heuristic assumes that the developers who were good helpers and able to assist in the past might be the developers who have the most ability to help with the current code. Developers are considered more social as the number of times they have helped others increases.

iv. How Often Developers are Recommended: Following Begel et al. [4], we provide the ability for developers to recommend other developers, and then we use these recommendations to improve our system recommendations to the Active Developer. We also use these recommendations to detect the developer reputations within the ERSF. We consider that the more developers have been recommended by others, the better the reputation is of these developers, so they might be good developers to recommend as helpers. Reputation is

identified as “how much a developer was recommended to help by others in the past within the ERSF”.

3.3.2. Experts Recommender System Design

Our ERSF approach is designed on top of the SimCad Clone Detection tool and the Eclipse Communication Framework/DocShare plugin (ECF/DocShare). The SimCad Clone Detection tool is used in our work to extract code fragments similar to the one at hand, as we mentioned in the technical heuristics identifications. On the other hand, the ECF/DocShare plugin is used to provide a channel to the developers to communicate with the recommended experts to get assistance. These communications are then tracked by our system and are saved in the RS MySQL database in order to design the social heuristics measurements for the improvement of the system performance and recommendations. In this section, we first provide background of the SimCad tool followed by background of the ECF/DocShare plugin. Then, we explain our architecture of the ERSF and show how the SimCad and the ECF/DocShare are included, in it.

3.3.2.1. SimCad Clone Detection Tool Background

The SimCad detects similar code fragments, which are called clone fragments, to the one at hand. Clone research classifies these clones into four different types (Type-1, Type-2, Type-3, and Type-4) according to their degree of similarities, as explained below [34]:

- i. Type-1 Clone:** “Identical code fragments except for variations in white-spaces and comments.”
- ii. Type-2 Clone:** “Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments.”
- iii. Type-3 Clone:** “Code fragments that exhibit similarity as of Type-2 clones and also allow further differences such as additions, deletions or modifications statements.”

iv. Type-4 Clone: “Code fragments that exhibit identical functional behavior but implemented through very different syntactic structure”.

However, since the SimCad tool, which our recommender system is developed on top of it, does not detect Type-4 clone fragments, we also do not consider this type in our work.

3.3.2.2. Eclipse Communication Framework/DocShare (ECF/DocShare) Background

The ECF/DocShare plugin allows two developers in a distributed location to share their editors, so both can collaborate to write or modify the shared code. In addition, they can chat while they are sharing the editor.

When a developer (e.g. dev1) wants to share his/her editor, s/he right-clicks on the editor and then selects “share editor with” from the pop-up menu. All available developers are shown so s/he can click on the one s/he would like to share the editor with (e.g. dev2). Figure 3.3 shows an example of this right menu and how dev1 can set up the editor for sharing.

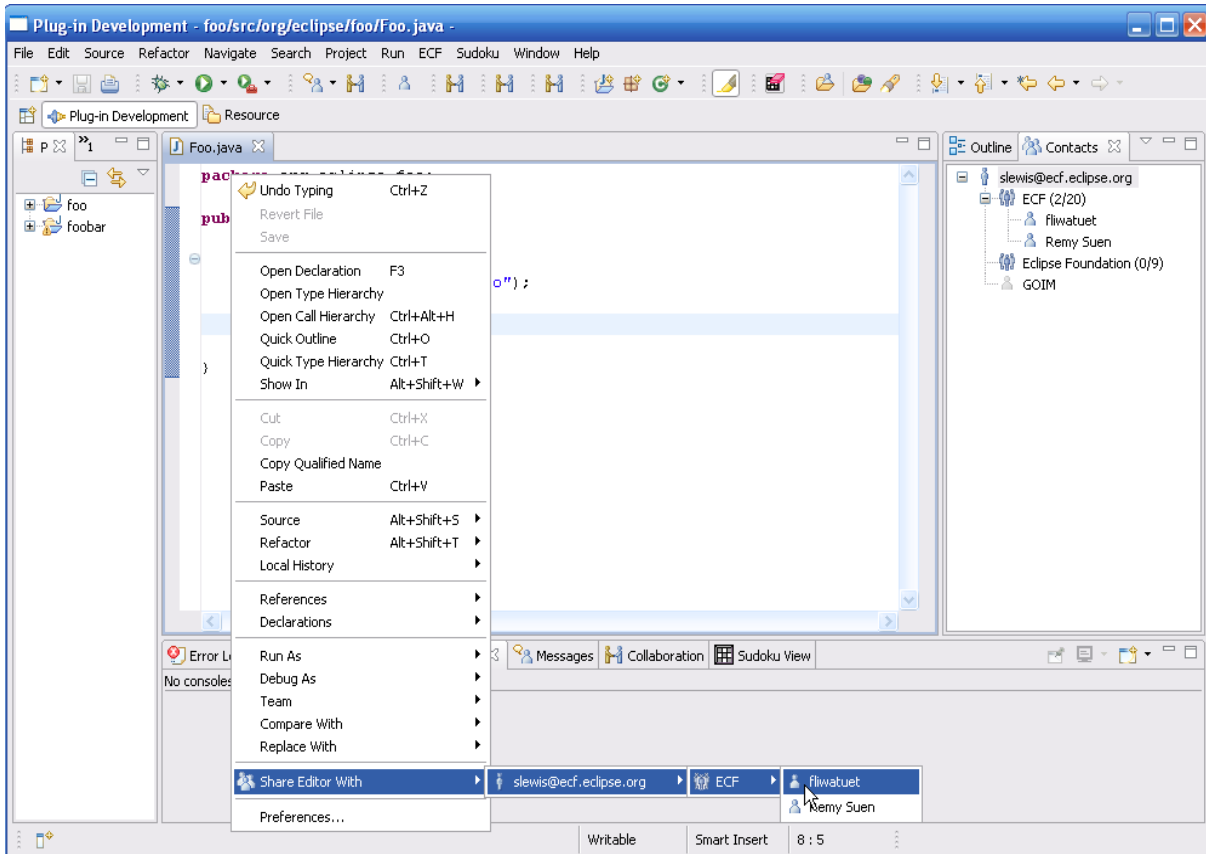


Figure 3.3: ECF/DocShare "Share Editor With" Menu

A pop-up window is then shown to dev2 asking whether s/he will accept this sharing as shown in Figure 3.4.

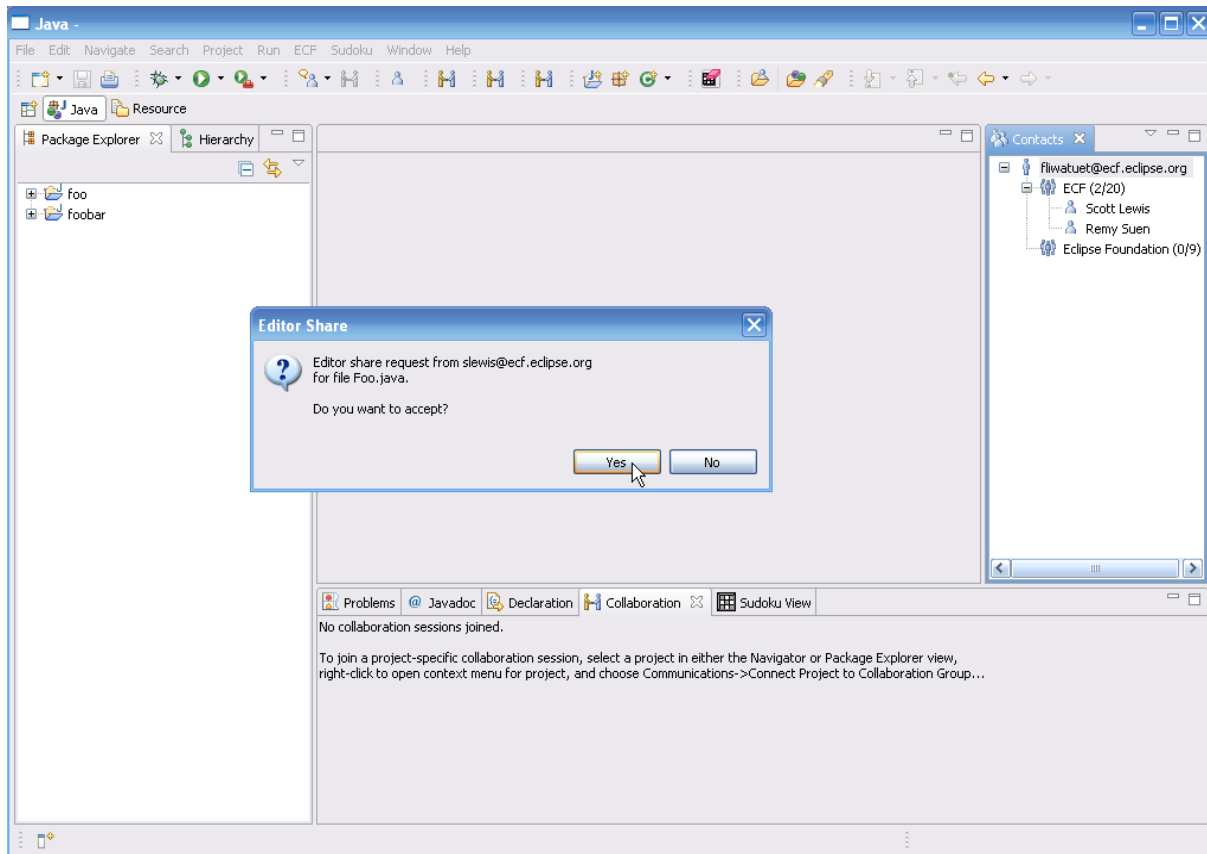


Figure 3.4: Sharing Permission Pop-up Window

If dev2 accepts the sharing request, dev1's editor is then opened in dev2's framework.

Figure 3.5 shows dev2's Eclipse framework in which dev1's editor is opened. The plugin allows both developers to work on the code and chat at the same time.

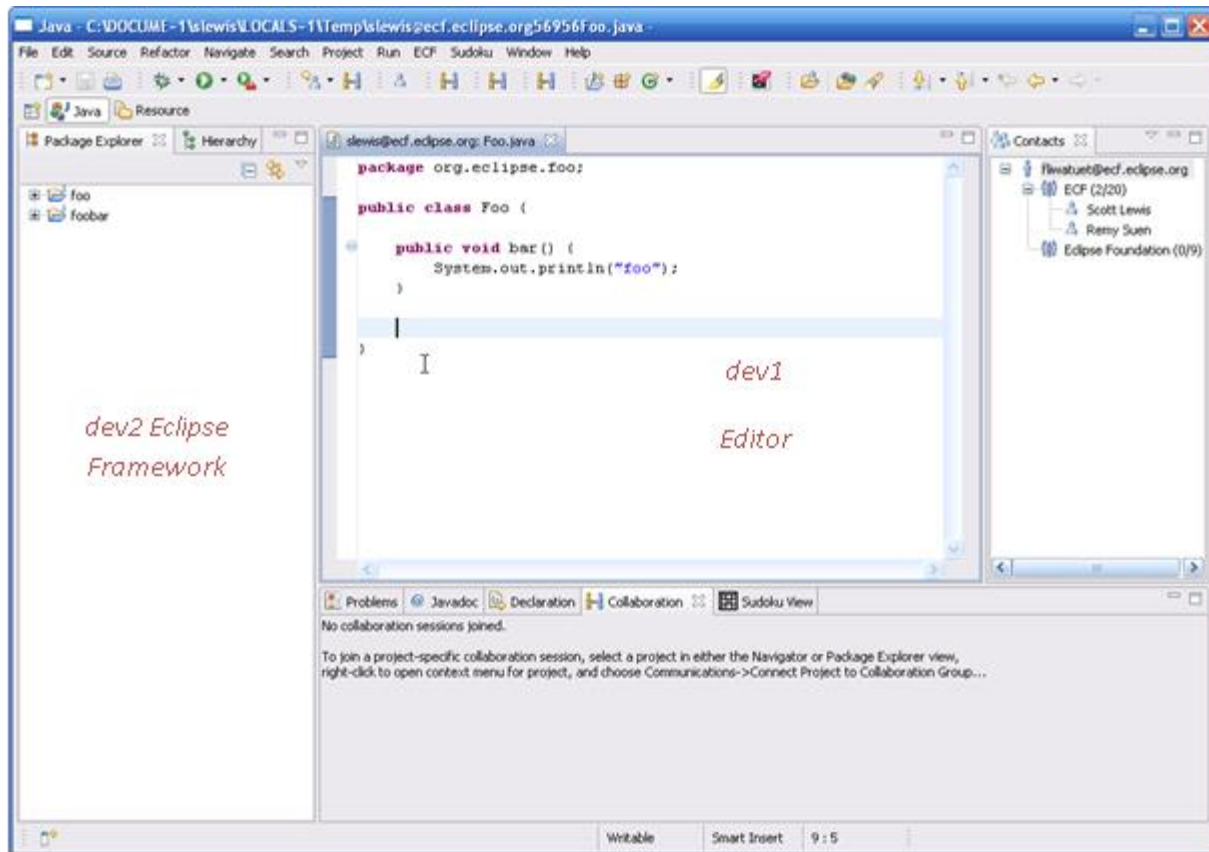


Figure 3.5: dev1's Editor Opened in dev2's Framework Example

Finally, dev1 can stop the sharing by selecting the “Stop Sharing Editor with...” option, which closes the shared editor in dev2’s framework.

3.3.2.3. Experts Recommender System Architecture

Figure 3.6 shows the architecture of the ERSF, what the main components of the system are, and how they are connected to each other and to the SimCad tool and the ECF/DocShare plugin.

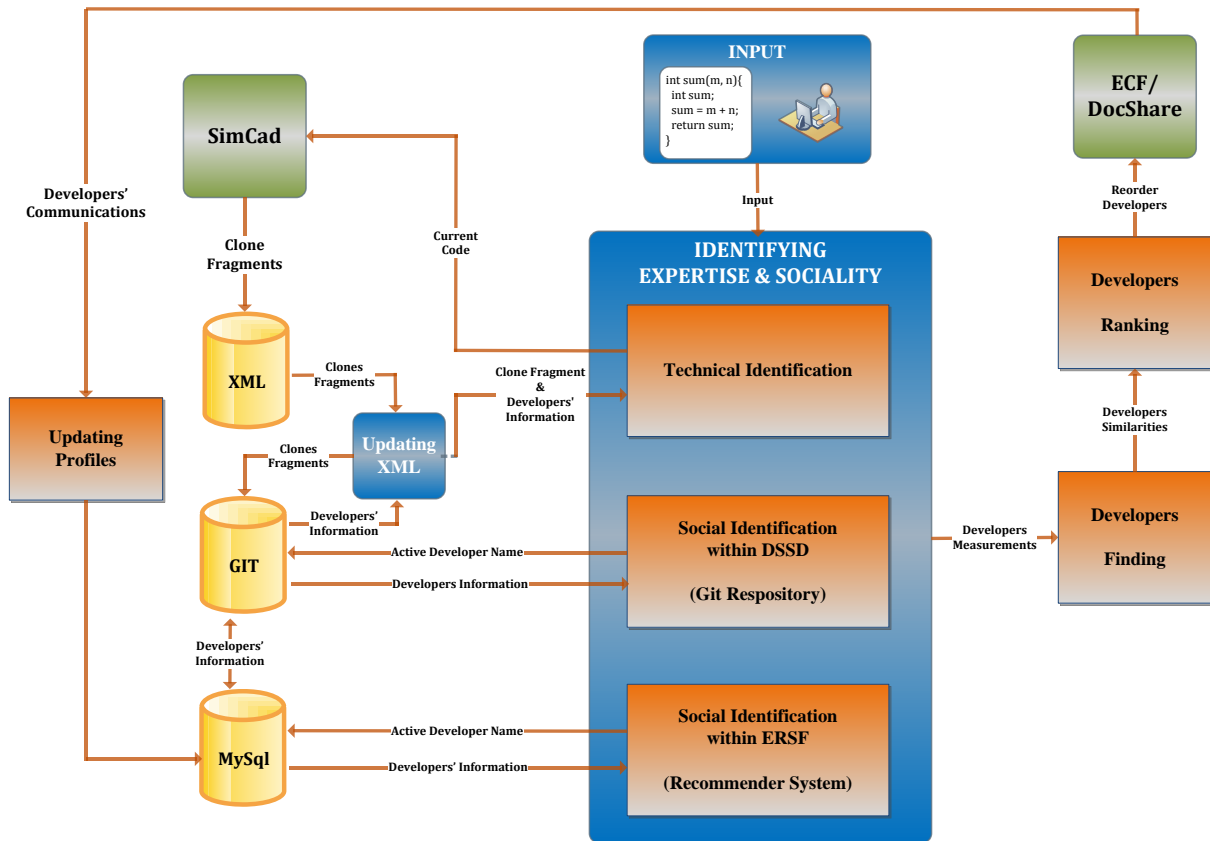


Figure 3.6: ERSF Architecture

When the Active Developer asks for help, his/her name and the code fragment in question are captured as input to the system. The system first identifies the expertise and sociality of each developer within the organization using the technical and social heuristics. Then, it finds who might be suitable experts to recommend and ranks them according to their likelihood to be good helpers, following the heuristics discussed in the last section. Finally, it recommends this ranked list of developers as experts to the Active Developer. Below we explain the main components that implement these functions.

A. Identifying Expertise and Sociality

Since experts in our system are identified using three different groups of heuristics (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF),

the identification in the system architecture is divided into three different components as well. Each of these components is responsible for one of these groups as explained below.

i. Technical Identification: The technical identification component sends the source code input to the SimCad tool, which finds the clone fragments of the given code and saves it in an XML file. This resulting file is then updated in the XML Updating component to include information about developers who have worked on those clone fragments, which is extracted from the Git repository. The technical identification component uses the updated XML file to analyze the developer expertise on the similar fragments including the input code fragment if that is not a new fragment (clone fragments) and measures their expertise using the technical heuristics.

ii. Social Identification within the DSSD: The social identification within the DSSD component takes the Active Developer name and extracts his/her communications with other developers. It also extracts the developer communications with each other. These data are extracted from the Git Repository used to design the social heuristics in order to measure the developer relationships to the Active Developer and their sociality within the DSSD.

iii. Social Identification within the ERSF: As with the previous component, this component takes the Active Developer name and extracts his/her communications with other developers, as well as the developer communications with each other. However, this component extracts the data from the RS MySQL database, which has the tracked communications through the ERSF. The purpose for this component is to measure the developer relationship with the Active Developer and their sociality as in the previous component but using the social heuristics that are measured using the communications through our ERSF.

B. Updating XML

The XML file from the SimCad includes the clone fragments grouped according to their types. Each fragment includes the file path, start line and end line of the fragment and is identified by its ID. The system in this component edits the XML file to include information about developers who have worked on each fragment. For each line within the fragment, we extract its commit ID, author name, last date of modifications, and the line number from the Git repository and add them as new elements under this clone fragment. The author data are then used for the technical heuristics.

C. Finding Experts

After the system measures the developer expertise in the clone fragments using the technical heuristics, the developer relationships to the Active Developer, their sociality within the DSSD and the ERSF using the social heuristics, the system takes these measurements to find the developers who might be suitable experts to recommend to help the Active Developer to complete the code at hand. We weight each of these heuristics based on their importance in designing our algorithm. Then, we apply the algorithm to each developer to measure their likelihood to be the suitable experts to recommend to help the Active Developer.

D. Ranking Developers

The main goal of this system is to recommend a ranked list of developers. Therefore, this component reorders the developers in the ECF/DocShare right-click menu (Figure 3.3) to display a ranked list of developers according to their likelihoods from the previous component to be suitable experts to help the Active Developer.

E. Updating Profiles

Through the ECF/DocShare plugin and the ERSF, the Active Developer can contact a developer s/he would like to get help from. We have developed our ERSF to track this communication and save it in the RS MySQL database in order to keep the developer information up-to-date and use it to measure the social heuristics when any developer needs experts to contact.

3.3.3. Experts Recommender System Implementation

Figure 3.7 shows the UML class diagram of our ERSF. We will use this diagram in explaining the system implementation, and we will mention the used methods as the `Class_name.Method_name`.

3.3.3.1. Sources

We mentioned in the introduction of this section that we have three different resources for our data. Below we explain the kinds of operations we use to access and deal with these data.

A. Git Class

The main functionality of the Git class method is to extract the data we need from the Git repository as in the following:

- i. getBlameInfo Method:** This method receives the file path, start line, and end line of the given clone fragment and extracts the developer information from the Git repository. This information includes commit ID, author name, date, and line number of each line within the fragment.
- ii. getLogInfo Method:** We use this method to extract the data needed for the social activities measurements. The method extracts from the repository the commits information depending on the received command, which contains the desired data for the task.

B. XML Class

The methods in the XML class deal with the clone file resulting from the SimCad tool, as shown below:

- i. updateXMLFile Method:** The output file from the SimCad includes the clone fragments of the current code as shown in Figure 3.8.

```
-<clones fragmentType="function" cloneSetType="group" nfragments="883" ngroups="255">
  -<clone_group groupid="1" nfragments="5" type="Type-3">
    <clone_fragment file="/src/NAnt.MSBuild/VS2008/OrcasSolutionProvider.cs.ifdefed" startline="36" endline="49" pcid="1532"/>
    <clone_fragment file="/src/NAnt.VSNet/Rainier/SolutionProvider.cs.ifdefed" startline="35" endline="48" pcid="787"/>
    <clone_fragment file="/src/NAnt.VSNet/Everett/SolutionProvider.cs.ifdefed" startline="35" endline="46" pcid="688"/>
    <clone_fragment file="/src/NAnt.MSBuild/VS2010/RosarioSolutionProvider.cs.ifdefed" startline="36" endline="49" pcid="1517"/>
    <clone_fragment file="/src/NAnt.MSBuild/VS2005/WhidbeySolutionProvider.cs.ifdefed" startline="36" endline="49" pcid="1529"/>
  </clone_group>
  -<clone_group groupid="2" nfragments="34" type="Type-3">
    <clone_fragment file="/src/Tasks/Msi/InstallerCreationCommand.cs.ifdefed" startline="266" endline="274" pcid="1943"/>
    <clone_fragment file="/src/NAnt.Core/Tasks/LoopTask.cs.ifdefed" startline="440" endline="454" pcid="1264"/>
    <clone_fragment file="/src/Tasks/Msi/InstallerCreationCommand.cs.ifdefed" startline="2671" endline="2691" pcid="2014"/>
  </clone_group>
</clones>
```

Figure 3.8: XML File from SimCad

We need to analyze who has worked on those clones in order to get the technical measurements. Therefore, for each clone fragment, we extract the developer information by calling on the `Git.getBlameInfo` method. Then, we use the returned data to update the XML file, as shown in Figure 3.9.

```

- <clones fragmentType="function" cloneSetType="group" nfragments="883" ngroups="255">
- <clone_group groupId="1" nfragments="5" type="Type-3">
- <clone_fragment file="/src/NAnt.MSBuild/VS2008/OrcasSolutionProvider.cs.ifdefed" startline="36" endline="49" pcid="1532">
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="36"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="37"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="38"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="39"/>
  <blameInfo commitId="b470f641" author="rmboggs" date="2011-08-08" time="05:25:02" lineNo="40"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="41"/>
  <blameInfo commitId="b470f641" author="rmboggs" date="2011-08-08" time="05:25:02" lineNo="42"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="43"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="44"/>
  <blameInfo commitId="b470f641" author="rmboggs" date="2011-08-08" time="05:25:02" lineNo="45"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="46"/>
  <blameInfo commitId="b470f641" author="rmboggs" date="2011-08-08" time="05:25:02" lineNo="47"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="48"/>
  <blameInfo commitId="edb96521" author="drieseng" date="2006-12-22" time="17:30:25" lineNo="49"/>
</clone_fragment>
- <clone_fragment file="/src/NAnt.VSNet.Rainier/SolutionProvider.cs.ifdefed" startline="35" endline="48" pcid="787">
  <blameInfo commitId="1c6f1b2e" author="drieseng" date="2006-12-22" time="17:35:00" lineNo="35"/>
  <blameInfo commitId="1c6f1b2e" author="drieseng" date="2006-12-22" time="17:35:00" lineNo="36"/>
  <blameInfo commitId="1c6f1b2e" author="drieseng" date="2006-12-22" time="17:35:00" lineNo="37"/>
  <blameInfo commitId="1c6f1b2e" author="drieseng" date="2006-12-22" time="17:35:00" lineNo="38"/>

```

Figure 3.9: Updated XML File with Developers' Information

ii. `getDevelopersTechnicalInfo` Method: The method returns the clone and developer information from the updated XML file.

C. `RS_MySql` Class

This class is used to access and add, modify, or get data from the RS MySQL database.

Before we explain the method functionality in this class, we describe the database and its tables.

a) `RS Database`

Figure 3.10 shows the structure of the RS database and its tables, as described below:

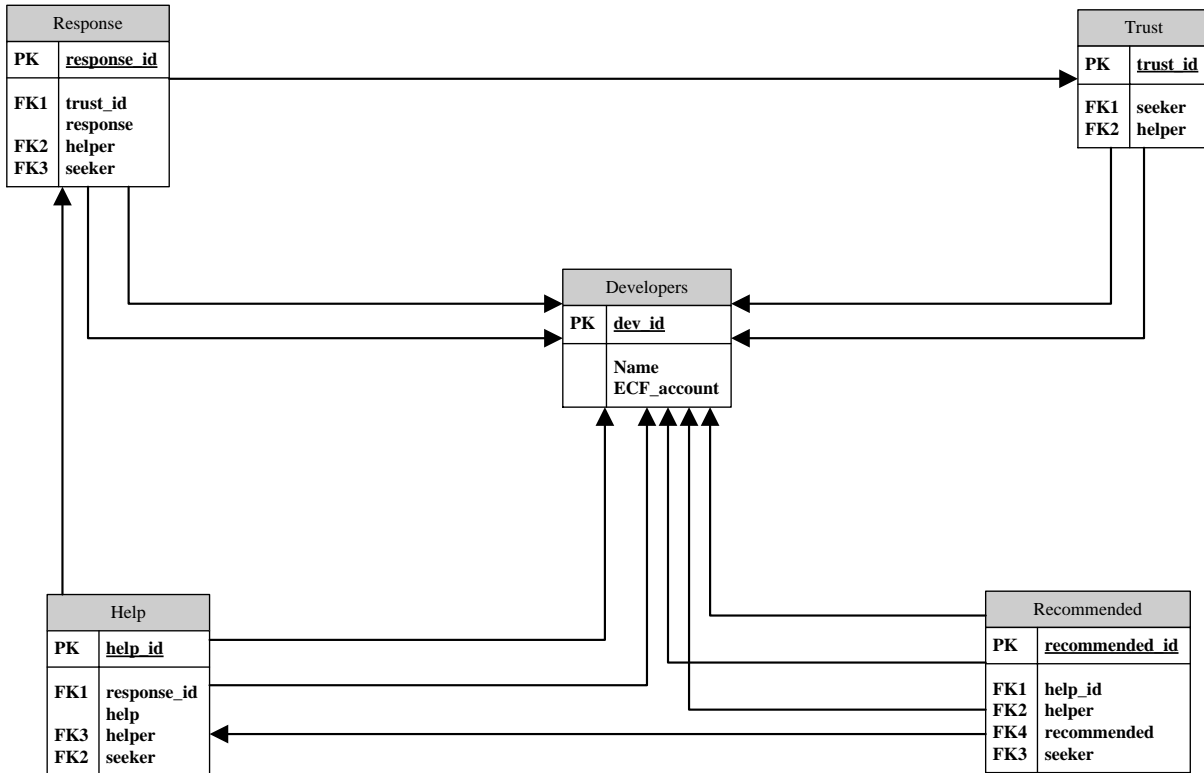


Figure 3.10: RS MySQL Database

- i. Developers Table:** This table has all the developers within the DSSD and the emails they are logged in within the ECF plug-in “ECF_email”.
- ii. Trust Table:** This includes all the trusts between the developers, which is represented by the “helper”s that “seeker”s have trusted.
- iii. Response Table:** This table shows if the helper has responded to the seeker request in the past (“response = 0” if s/he did not respond or “response = 1” if s/he responded).
- iv. Help Table:** Like the Response Table, the Help Table represents whether the “helper” was able or not able to help the “seeker” (“help = 1” or “help = 0”, respectively).
- v. Recommended Table:** represents whom has been “recommended” to the “seeker” by the “helper”.

b) RS_MySql Class Methods

Below are the methods that are used to deal with the RS MySql database:

- i. insertTrust Method:** This method inserts a new record in the Trust table.
- ii. insertResponse Method:** This inserts a new record in the Response table.
- iii. insertHelp Method:** This inserts a new record in the Help table.
- iv. insertRecommended Method:** This inserts a new record in the Recommended table.
- v. AD_getTrust Method:** This receives the name of the “seeker” and returns information of the “helper” whom the “seeker” has trusted in the past.
- vi. AD_getResponse Method:** This receives the name of the “seeker” and returns all the “helper”s who have responded to this “seeker” in the past.
- vii. AD_getHelp Method:** This receives the name of the “seeker” and returns all the “helper”s who were able to help the “seeker” in the past.
- viii. AD_getRecommended Method:** This receives the name of the “seeker” and returns all the “recommended” developers who have been recommended to this “seeker” in the past.
- ix. RSF_getTrust Method:** This returns all the “helper”s who were trusted by others in the past.
- x. RSF_getResponse Method:** This returns all the “helper”s who have responded to the help requests in the past.
- xi. RSF_getHelp Method:** This returns all the “helper”s who were able to help the developer in the past.
- xii. RSF_getRecommended:** This returns all the “recommended” developers who have been recommended by others to help in the past.

3.3.3.2. Identifying Expertise and Sociality

In this section we explain how we implement the technical heuristics to measure the developer technical expertise as well as how to implement the social heuristics within both DSSD and ERSF to measure the developer sociality.

A. TechnicalExpertise Class

We explained in the technical identifications that we use four different heuristics to measure the developer technical expertise. Each of these heuristics has an implemented method as follows:

- i. cloneTypes Method:** This method gets the clone fragments and their developers' information by calling the XML.getDevelopersTechnicalData method. It then uses this information to determine which type(s) of clone each developer has cooperated on.
- ii. noOfFragments Method:** This gets the clone fragments and their developers' information by calling the XML.getDevelopersTechnicalData method. It then uses this information to count the number of fragments each developer has worked on.
- iii. noOfLines Method:** This gets the clone fragments and their developers' information by calling the XML.getDevelopersTechnicalData method. It then uses this information to count how many lines each developer has written or modified.
- iv. mostRecentlyModification Method:** This gets the clone fragments and their developers' information by calling the XML.getDevelopersTechnicalData method. It then uses this information to determine when the last time was the each developer cooperated on those clones.

B. SocialActivites Class

This class has the methods that implement the social heuristics within the DSSD as follows:

- i. AD_noOfSharedFiles Method:** This gets the commits information from the Git repository using the `Git.getLogInfo` method. Then, it analyzes how many files each developer has cooperated on with the Active Developer regarding their creation/modification.
- ii. AD_noOfSharedCommits Method:** This gets the commits information from the Git repository using the `Git.getLogInfo` method. Then, it analyzes how many commits each developer has collaborated on with the Active Developer either as authors or committers.
- iii. DSSD_noOfSharedFiles Method:** This gets the commits information from the Git repository using the `Git.getLogInfo` method. Then, it counts for each developer within the DSSD the number of files they have cooperated on with other developers regarding their creation/modification.
- iv. DSSD_noOfSharedCommits Method:** This gets the commits information from the Git repository using the `Git.getLogInfo` method. Then, it counts for each developer within the DSSD the number of commits they have shared with other developers either as authors or committers.

C. RS_SocialActivities Class

The methods in this class implement the social heuristics within our ERSF. Each of the following methods implements one social heuristic.

- i. AD_trust Method:** This method sends the Active Developer name to the `RS_MySql.AD_getTrust` method to get all the developers s/he has trusted in the past. After that, it counts the number of times each developer has been trusted by the Active Developer.
- ii. AD_response Method:** This sends the Active Developer name to the `RS_MySql.AD_getResponse` method to get all the developers who have responded to the

Active Developer. After that, it counts how many times each developer has responded to the Active Developer requests in the past.

iii. AD_help Method: This sends the Active Developer name to the RS_MySql.AD_getHelp method to get all the developers who were able to help the Active Developer in the past.

After that, it counts how many times each developer was able to help the Active Developer.

iv. AD_recommended Method: This sends the Active Developer name to the RS_MySql.AD_getRecommended method to get all the developers who were recommended to him/her in the past. Then, it counts how many times each developer has been recommended to this Active Developer by others.

v. RSF_trust Method: This calls the RS_MySql.RSF_getTrust method and counts for each developer how many times s/he has been trusted by others in the past.

vi. RSF_response Method: This calls the RS_MySql.RSF_getResponse method and counts for each developer how many times s/he has responded to others in the past.

vii. RSF_help Method: This calls the RS_MySql.RSF_getHelp method and counts for each developer how many times s/he was able to help others in the past.

viii. RSF_recommended Method: This calls the RS_MySql.RSF_getRecommended method and counts for each developer how many times s/he has been recommended by others to help in the past.

3.3.3.3. Finding and Ranking Experts

A. FindingAndRankingExperts Class

This class is responsible for computing the developer likelihood to be an expert and then ranks those experts based on their likelihoods to be recommended to help the Active Developer.

i. expertiseSocialityComputing Method: The method computes based on the knowledge in the code at hand and based on the relationship with the Active Developer and sociality within the organization (either within the DSSD or ERSF), the expertise and sociality of each developer within the organization using a combination of all the heuristics we explained in Identifying Expert section using the following formula (Formula-1):

$$D_e = \sum_{h=1}^n gw \left(\frac{D_{(s/e)}}{T_{(s/e)}} \times hw \right) \quad 0 \leq D_e \leq 1 \quad \text{Formula-1}$$

Where D_e is the current developer for whom we are computing his/her likelihood to be an expert, h is the current heuristic the ratio is computed under, n is the total number of heuristics, gw is the group weight where this heuristic is classified under (technical heuristic, social heuristic within the DSSD, or social heuristic within the ERSF), $D_{(e/s)}$ is the current developer expertise/sociality under this heuristic, $T_{(e/s)}$ is the total expertise/sociality under this heuristic, and hw is the heuristic weight.

The algorithm is developed first based on one technical/social heuristic. Thus, to find the developer expertise/sociality under this heuristic, we compute the ratio of his/her expertise/sociality relative to other developers' expertise/sociality under this heuristic using formula-2:

$$D_e = \frac{D_{(e/s)}}{T_{(e/s)}} \quad \text{Formula-2}$$

For example, if we consider the *Number of Lines* heuristic, and we assume that we have a piece of code with 15 lines (T_e). Three developers $D1$, $D2$, and $D3$ have collaborated on the modification of this code as follows: $D1$ has written 4 lines, $D2$ has written 8 lines, and $D3$

has written 3 lines out of 15. We then would like to compute the likelihood of each of the three developers to be an expert using the *Number of Lines* heuristic, so we will apply the above formula (Formula -2) on each developer as shown in Table 3.1:

Table 3.1: Developer Likelihood to be an Expert Example (Formula-2)

Developers	D1	D2	D3	T_(e)
Heuristics				
Number of Lines	4	8	3	15
D_e	4/15 = 0.27	8/15 = 0.53	3/15 = 0.2	-

However, since we have more than one heuristic that need to be considered in computing the likelihood of a developer to be an expert, we combined these heuristics in the algorithm by finding the summations of their ratios for this particular developer with formula-3.

$$D_e = \sum_{h=1}^n \frac{D_{(e/s)}}{T_{(e/s)}} \quad \text{Formula-3}$$

For instance, we assume that we have a *Trust* heuristic; which includes 10 trusts in total between the above developers (*D1*, *D2*, and *D3*), besides the *Number of Lines* heuristic. Table 3.2 shows the expertise and sociality of the three developers (*D1*, *D2*, and *D3*) and the two heuristics (*Number of Lines* and *Trust*) that are considered to compute the likelihood of each of the three developers to be an expert; the last row shows how we apply formula-3 for each developer.

Table 3.2: Developer Likelihood to be an Expert Example (Formula-3)

Developers Heuristics	D1	D2	D3	T_h
Number of Lines	4	8	3	15
Trust	5	2	3	10
D_e	4/15 + 5/10 = 0.77	8/15 + 2/10 = 0.73	3/15 + 3/10 = 0.5	-

Another important aspect of having more than one heuristic in the algorithm is that not all the heuristics within a group (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF) have the same weights since not all of them have the same priorities and importance in recommending experts. Therefore, we have worked on determining those priorities based on human judge decisions. We have conducted an experiment to analyze those priorities. In the experiment, we gave the judges a list of developers with their expertise and sociality that are represented by the technical and social heuristics, and we asked them to rank the first three developers and select the heuristics they have considered while they were ranking the developers. After that, we used the Weka tool to analyze the judge rankings and come up with the heuristic weights within a particular group (this is explained in detail in Chapter 4). Based on this, we developed our algorithm to consider the heuristics weights (hw) as in formula-4:

$$D_e = \sum_{h=1}^n \left(\frac{D_{(s/e)}}{T_{(s/e)}} \times hw \right) \quad \text{Formula-4}$$

Moreover, since we have three groups, it is also desired to analyze the priorities and importance of each group between them. Thus, we used both the judge rankings and the

Weka tool as well to determine each group weights (gw) based on its importance. As a result, we improved our algorithm as in formula-5; which is the complete version that is used in recommending experts.

$$D_e = \sum_{h=1}^n gw \left(\frac{D_{(e/h)}}{T_{(e/h)}} \times hw \right) \quad 0 \leq D_e \leq 1 \quad \text{Formula-5}$$

Where D_e is the current developer for whom we are computing his/her likelihood to be an expert, h is the current heuristic the ratio is computed under, n is the total number of heuristics, gw is the group weight where this heuristic is classified under (technical heuristic, social heuristic within DSSD, or social heuristic within ERSF), $D_{(e/h)}$ is the current developer expertise/ sociality under this heuristic, $T_{(e/h)}$ is the total expertise/ sociality under this heuristic, and hw is the heuristic weight.

The full experiment with the human judge rankings and the resulting weights of the heuristics and the groups that we used in designing our algorithm is explained in Chapter 4.

ii. expertsRanking Method: This method takes the list of developers with their likelihoods to be experts from the previous method and ranks them in descending order as who are the best experts to assist the Active Developer. In our example, the ranking list is D1, D2, and D3.

3.3.3.4. Updating Profiles

i. updatingProfiles Class

We showed in the Design section that our recommender system approach is implemented on top of the ECF. Thus, the methods in this class capture the developer communications within the ECF and the ERSF and update the RS MySQL database in order to keep the developer profiles up-to-date for future recommendations.

- ii. captureTrust:** When the Active Developer picks a developer from the ECF menu (Figure 3.3) to contact and get help, this method captures this selection as a trust value from the Active Developer to the selected developer. Then, it sends this trust to the RS_MySql.insertTrust method to be saved as a new trust in the RS_MySql database.
- iii. captureResponse:** The response in this method is captured when a pop-up window is shown to the selected developer (Figure 3.4). This method gets the developer reaction and sends it to the RS_MySql.insertResponse method to be saved as a new response.
- iv. captureHelp:** When the Active Developer stops the editor sharing, our system displays a pop-up window asking if the selected developer was able to help complete the code at hand. Then, this method calls the RS_MySql.insertHelp method to add the captured reaction to the RS MySal database.
- v. captureRecommended:** If the selected developer could not help the Active Developer, s/he can recommend another developer to help. This method captures who is recommended and calls the RS_MySql.insertRecommended method to add this recommendation.

CHAPTER 4

EXPERIMENT AND EVALUATION

4.1. INTRODUCTION

In our work we came up with 16 heuristics to measure the developer expertise and sociality, which we classified into three different groups (four technical heuristics, four social heuristics within the DSSD, and eight social heuristics within the ERSF) as we explained in Chapter 3.

However, not all of these heuristics have the same degree of importance. Some heuristics might be given the highest priority in measuring the expertise or sociality, some might be given less priority, some might be dependent on others, and some of these heuristics should be omitted in finding the experts. Moreover, the heuristic groups themselves also do not have the same importance when compared to each other. A group might be substantial in identifying the experts, might be less important but also should be considered, or might lose or gain its importance as the recommender system is run.

These issues were analyzed and understood based on analyzing human judgements. The importance of the heuristics and the groups, which is represented with numeric weights, was then used to design the recommender system algorithm. This algorithm was then evaluated using the human judgements, as well. Besides, we applied NiveBayes, NaiveNet, and J48 [29] machine learning algorithms on the human rankings, and then we also evaluated our algorithm based on the predicted rankings by these machine learning algorithms.

In this chapter, we explain how we conducted our experiment, how we determined the heuristic and the group weights according to their importance, how we evaluated our algorithm in recommending the experts, and the results of this experiment.

The chapter is organized as follow: Section 4.2 provides an overview of the experiment, Section 4.3 explains the experiment methodology, and Section 4.4 discusses the results.

4.2. EXPERIMENT OVERVIEW

Our experiment went through four phases. The first phase was concerned with collecting the human rankings. The second phase analyzed these rankings in order to determine the weights of both the heuristics and the groups. The third phase used these weights in order to design the recommendation algorithm. The last phase was concerned with evaluating the accuracy of the algorithm in recommending the experts. Figure 4.1 depicts an overview of our experiment and the four phases.

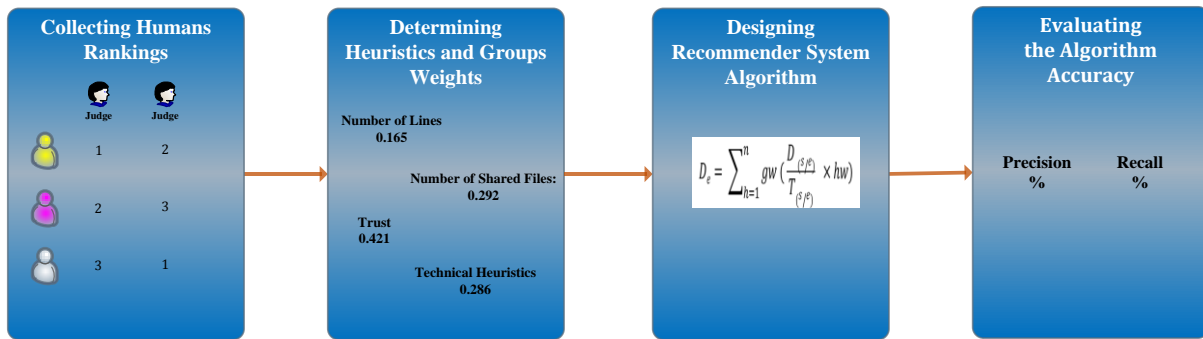


Figure 4.1: Four Phases of the Experiment

Collecting Human Rankings Phase

Our target in this phase was developers (we call them judges to distinguish them from the "developers" for whom we built the system). We gave the judges three different scenarios with a list of developers and some data representing their expertise and sociality. Then, we asked the judges to rank the top three developers that they thought were the best to contact and get help from; also, we asked them to select which of the technical and/or social heuristics they considered while they ranking each of the three developers.

Determining Heuristic and Group Weights Phase

In this phase, we analyzed the heuristics that the judges considered in the previous phase while they were ranking the first three developers in order to determine their importance and priorities from a human perspective. This importance was then analyzed and represented with numeric weights. We also analyzed the groups under which these heuristics were classified to weight them according to their importance between each other.

Designing Recommender System Algorithm Phase

The weights of both the heuristics and the groups from the second phase were then used to design our algorithm for recommending experts in this phase.

Evaluating the Algorithm Accuracy Phase

This phase is concerned with evaluating the accuracy of our algorithm in recommending the experts. This is done by comparing our algorithm rankings to the judge rankings as well as to the rankings of machine learning algorithms.

4.3. EXPERIMENT METHODOLOGY

In the previous section, we provided an overview of our experiment with brief explanations of each phase within this experiment. In this section we explain how we implemented each of the phases in detail.

4.3.1. Collecting Human Rankings Phase

Our experiment was built based on human judgments. We ran the experiment using 10 judges who were artificial intelligence, software engineering, and social computing graduate students from the University of Saskatchewan.

We provided the judges with a list of developers and some data representing their expertise and/or sociality. However, since we have a large number of heuristics in our algorithm, we

composed three scenarios, each concerned with a group of heuristics (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF). In addition, we wanted the judges to consider the heuristic group of the previous scenario while they were making their decision in the current one, so we included the data from that previous scenario to be considered in the current one. For instance, if the judges were working on Scenario-2, which is concerned with the social heuristics within the DSSD, we still wanted them to consider the technical group, which is the concern of Scenario-1; therefore, we also included the data describing the technical expertise of a developer in Scenario-2.

While the study was running, we asked our judges to assume that they were the Active Developer who was looking for experts for help. Then, we started each scenario with a brief explanation of what it was about, and we gave the judges a list of developers with some data representing the developers' characteristics depending on the scenario they were working on, and we asked the judges to rank the first, second, and the third developers that they thought were the best experts to contact and get help from. Also, we asked them to choose the reasons for their selections; the reasons they could select from were the same heuristics we suggested for our algorithm.

The human rankings results include three parts: the ranked developers by the judges, the considered heuristics, and the given rankings (i.e. 1, 2, or 3) to these developers. These parts are then used as follow:

1. Both the considered heuristics and the given rankings were analyzed to find out the heuristics and group weights for designing the recommender system algorithm (section 4.3.2)

2. The three parts together (the ranked developers by the judges, the considered heuristics, and the given rankings to these developers) were used to evaluate the accuracy of the recommender system algorithm (section 4.3.4).

Below we explain each of the scenarios in detail:

4.3.1.1. Scenario-1

Scenario-1 studies the technical heuristics. We gave the judges a piece of code supposing it was the one they need help on (Figure 4.2) and five similar/clone fragments of that code: two of Type-1 (Figure 4.3), two of Type-2 (Figure 4.4), and one of Type-3 (Figure 4.5) with the names of the developers who modified these fragments and the date of the last modifications.

Scenario 1

Scenario 1 studies the developers' knowledge and expertise on source code. We extract similar code fragments, which are called Clones, to the one the Active Developer needs help on and extract who have cooperated on those fragments. We assume that the developers who have cooperated on those clones might understand and help on the code at hand better than other developers in the organization.

However, degree of similarities of these clones (clone types) to the one at hand might differ; thus, that might affect the developers' expertise as well. Moreover, this expertise might be affected by the number of fragments, the number of lines, and the date of the last modification each developer has done.

Now, you are working on the following code, and you do not know how to complete it. So, you are looking for experts to help.

```
private string FindParent(string DirectoryName, MSIDirectory directory) {
    foreach (MSIDirectory directory2 in directory.directory) {
        if (directory2.name == DirectoryName) {
            return directory.name;
        } else {
            string parent = FindParent(DirectoryName, directory2);

            if (parent != null) {
                return parent;
            }
        }
    }
    return null;
}
```

Figure 4.2: The Code Fragment the Active Developer Needs Help On

Type-1 Clone Fragments

The yellow highlight in below clone fragments shows the differentiations in the layout and whitespace from your code, and added comments.

private string FindParent (string DirectoryName,	Drieseng	2003-05-03
MSIDirectory directory) {	Drieseng	2003-05-03
foreach (MSIDirectory directory2 in directory.directory) {	Michael C. Two	2002-10-25
// Look in each directory	Michael C. Two	2002-10-25
if (directory2.name == DirectoryName) {	Michael C. Two	2002-10-25
return directory.name;	Michael C. Two	2002-10-25
} else	Michael C. Two	2002-10-25
{	Michael C. Two	2002-10-25
string parent = FindParent(DirectoryName, directory2);	Michael C. Two	2002-10-25
if (parent != null) {	Michael C. Two	2002-10-25
return parent; }}}	Drieseng	2003-05-03
return null; }	Michael C. Two	2002-10-25

private string FindParent(string DirectoryName, MSIDirectory	Drieseng	2003-09-15
directory) {	Drieseng	2003-09-15
foreach (MSIDirectory directory2 in directory.directory) {	Scott Hernandez	2003-12-08
if (directory2.name == DirectoryName) { return directory.name; }	Scott Hernandez	2003-12-08
else {	Scott Hernandez	2003-12-08
string parent = FindParent(DirectoryName, directory2);	Scott Hernandez	2003-12-08
if (parent != null) { return parent; }	Drieseng	2003-09-15
}}	Drieseng	2003-09-15
return null;	Drieseng	2003-09-15
}	Drieseng	2003-09-15

Figure 4.3: Type-1 Clone Fragments

Type-2 Clone Fragments

The following 2 code fragments have more differentiations comparing to your code besides the ones in the previous 2 clones. They also differ in the identifiers' names as shown in the green highlight.

private string FindParent (string DirectoryName2,	Ryan Boggs	2012-05-04
MSIDirectory directory)	Ryan Boggs	2012-05-04
{	Ryan Boggs	2012-05-04
foreach (MSIDirectory directory2 in directory.directory) {	Ryan Boggs	2012-05-04
// Look in each directory	Ryan Boggs	2012-05-04
if (directory2.name == DirectoryName2) {	Ryan Boggs	2012-05-04
return directory.name;	Ryan Boggs	2012-05-04
}	Ryan Boggs	2012-05-04
else	Ryan Boggs	2012-05-04
{	Ryan Boggs	2012-05-04
string parent = FindParent(DirectoryName2, directory2);	Ryan Boggs	2012-05-04
if (parent != null)	Ryan Boggs	2012-05-04
{	Ryan Boggs	2012-05-04
return parent;	Ryan Boggs	2012-05-04
}}}	Ryan Boggs	2012-05-04
return null;	Ryan Boggs	2012-05-04
}	Ryan Boggs	2012-05-04
private string FindParentExample	Scott Hernandez	2003-11-16
(string DirectoryName, MSIDirectory directory) {	Scott Hernandez	2003-11-16
foreach (MSIDirectory directory2 in directory.directory) {	Scott Hernandez	2003-11-16
if (DirectoryName == directory2.name) { return directory.name;	Scott Hernandez	2003-11-16
} else {	Scott Hernandez	2003-11-16
string parent = FindParentExample(DirectoryName, directory2);	Scott Hernandez	2003-11-16
	Scott Hernandez	2003-11-16
if (parent != null) {	Scott Hernandez	2003-11-16
return parent; }}}	Ryan Boggs	2012-01-21
return null;	Ryan Boggs	2012-01-21
}	Scott Hernandez	2003-11-16

Figure 4.4: Type-2 Clone Fragments

Type-3 Clone Fragment

Besides the previous differentiations in Type-1 and Type-2, below fragment has added, removed, or changed statements. The 4 highlighted lines with blue are new lines that are not in your code.

<code>// Find the parent directory</code>	Drieseng	2003-05-26
<code>private string FindParent (string DirectoryName2,</code>	Gerry Shaw	2002-08-14
<code>MSIDirectory directory) {</code>	Gerry Shaw	2002-08-14
<code>if (DirectoryName == directory.name && directory is MSIRootDirectory) {</code>	Gerry Shaw	2002-08-14
<code>return ((MSIRootDirectory)directory).root;</code>	Gerry Shaw	2002-08-14
<code>} else {</code>	Gerry Shaw	2002-08-14
<code>if (directory.directory != null) {</code>	Gerry Shaw	2002-08-14
<code>foreach (MSIDirectory directory2 in directory.directory) {</code>	Gerry Shaw	2002-08-14
<code>// Look in each directory</code>	Gerry Shaw	2002-08-14
<code>if (directory2.name == DirectoryName2) {</code>	Gerry Shaw	2002-08-14
<code>return directory.name;</code>	Drieseng	2004-08-11
<code>} else</code>	Drieseng	2004-05-24
<code>{</code>	Drieseng	2004-08-11
<code>string parent = FindParent(DirectoryName2, directory2);</code>	Gerry Shaw	2002-08-14
<code>if (parent != null) {</code>	Gerry Shaw	2002-08-14
<code>return parent;</code>	Drieseng	2004-08-11
<code>}}}</code>	Drieseng	2004-08-11
<code>return null;</code>	Drieseng	2004-08-11
<code>}</code>	Drieseng	2004-08-11
	Gerry Shaw	2002-08-14

Figure 4.5: Type-3 Clone Fragment

To assist the judges in making their decisions, we summarized the developer technical expertise (degree of similarities/clone types, number of fragments, number of lines, and most recent modifications) on those clone fragments in a table (Figure 4.6). Each row in the table shows a developer expertise, and each column represents a technical characteristic. For instance, “Drieseng” is a developer who has worked on two fragments of the Type-1 clone; he did the most recent modifications of 3 lines on “2003/05/03” in the first fragment and the most recent modifications of 6 lines on “2003/09/15” in the second fragment. Also, he did the most recent modifications of 8 lines on “2004/11/08” in a Type-3 clone fragment. Appendix A contains a full list of developers with their technical expertise.

Summary of Developers' Cooperation

In the table, we summarize the developers' cooperation on the previous clone fragments. The table shows what clone type(s), how many fragment(s), how many line(s), and what is the date of last modification each of the developers has done.






Developer Name		Type-1		Type-2		Type-3	
		Number of Lines	date of Last Modification	Number of Lines	date of Last Modification	Number of Lines	date of Last Modification
	Drieseng	Fragment 1: 3 Fragment 2: 6	Fragment 1: 2003-05-03 Fragment 2: 2003-09-15	---	---	8	2004-08-11
	Michael C. Two	9	2002-10-25	---	---	---	---
	Scott Hernandez	4	2003-12-08	9	2003-11-16	---	---
	Ryan Boggs	---	---	Fragment 1: 17 Fragment 2: 2	Fragment 1: 2012-05-04 Fragment 2: 2012-01-21	---	---
	Gerry Shaw	---	---	---	---	12	2002-08-14

Figure 4.6: Developer Technical Expertise Summary

Finally, we asked the judges to rank the first three developers that they thought had the best expertise and who should be asked for help. Also, we asked the judges to select the reasons for their selections; these reasons represent the technical heuristics we are concerned with. Moreover, we also provided a text box if the judges had other reasons for their selections (Figure 4.7).

Developer Ranking

From the previous information of clones and developers, we would like you to rank those developers based on whom you will pick to get help from, and why you think this developer is the best one to contact and get help from.

First Developer

Select First Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Clones Types

- Has worked on Type-1
- Has worked on Type-2
- Has worked on Type-3

Other Factors

- Has worked on more than one Fragments
- Has modified a large number of lines
- Has modified code most recently

Any Other Comments

Others

Any Other Comments

If you Select "Others", Please Enter your Comments here

Second Developer

Select Second Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Clones Types

- Has worked on Type-1
- Has worked on Type-2
- Has worked on Type-3

Other Factors

- Has worked on more than one Fragments
- Has modified a large number of lines
- Has modified code most recently

Any Other Comments

Others

Any Other Comments

If you Select "Others", Please Enter your Comments here

Third Developer

Select Third Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Clones Types

- Has worked on Type-1
- Has worked on Type-2
- Has worked on Type-3

Other Factors

- Has worked on more than one Fragments
- Has modified a large number of lines
- Has modified code most recently

Any Other Comments

Others

Any Other Comments

If you Select "Others", Please Enter your Comments here

Next Scenario

Figure 4.7: Scenario-1 Developer Rankings

4.3.1.2. Scenario-2

Scenario-2 is concerned with the social heuristics within the DSSD: shared files and shared commits heuristics with the Active Developer as well as shared files and shared commits heuristics within the DSSD. We gave the judges in this scenario a list of developers with their social relationships and activities that represent the four heuristics; in addition, we also displayed the data related to the technical heuristics from scenario-1 and which developers were selected, which were marked by the yellow highlight to assist the judges in knowing who they selected, so

they do not face difficulty in remembering these experts (Figure 4.8). Appendix A contains a full list of developers with sociality within the DSSD.

Scenario 2

Scenario 2 studies how much each developer in the organization is close to the Active Developer by analyzing who have collaborated with him/her in the past. It considers that the developers who have cooperated with the Active developer in the past might be better experts to be recommended to help more than others. This closeness is measured by analyzing how many files a developer has collaborated on their creations or modifications with the Active Developer. Also, it is measured by analyzing how many commits a developer has collaborated in their submissions with the Active Developer.

Moreover, scenario 2 considers that a social developer within the organization might be a good expert who can help the Active Developer even if there is no any relationship between each other. Therefore, scenario 2 also studies the developers' sociality by analyzing how many files they have collaborated on with others. As well as, it analyzes how many commits a developer has cooperated on their submissions with others.

Below table shows the closeness of the developers to you with "Number of Files you Both have Cooperated on" and "Number of Commits you Both have Submitted". Also, it shows their sociality within the organization with "Number of Files a Developer has Cooperated on the Modification" and "Number of Commits a Developer has Cooperated on the Submissions". Moreover, it shows who have cooperated in the clones fragments from Scenario 1 with highlighting whom you selected to contact.

Developer Name		Social Activity with You				Social Activity within the Organization			
		Number of Files you Both have Cooperated on		Number of Commits you Both have Submitted		Number of Files a Developer has Cooperated on the Modification		Number of Commits a Developer has Cooperated on the Submissions	
Photo	Ryan Boggs	0		0		0		0	
		cloneTypes	Type-2	noOfFragments	2	noOfLines	19	dateOfLastModified	2012-05-04
Photo	Dguder	0		0		2		6	
Photo	Rmboggs	1		2		2		4	
Photo	Charles Chan	10		23		10		23	

Figure 4.8: Developer Social Heuristics within DSSD

At the end, the judges were asked to rank the first three developers they thought were the best to assist them. Also, they were given the four social heuristics as reasons for their selection and a "Has worked on similar code fragments (Scenario-1)" option to select if they chose the developers because they had technical expertise on any of clone fragment(s) (Figure 4.9).

Developer Ranking

From the previous information of the developers' closeness to you and their sociality within the organization, we would like you to rank those developers based on whom you will pick to get help from, and why you think this developer is the best one to contact and get help from.

First Developer

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Any Other Comments

Select First Developer

Technical Factors
Has worked on Clones (Scenario 1)

Social Factors

Social Activity with You
Has Collaborated with Me on Many Files
Has Submitted Many Commits with Me

Social Activity within the Organization
Has Collaborated on Many Files with Others
Has Submitted Many Commits with Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Second Developer

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Any Other Comments

Select Second Developer

Technical Factors
Has worked on Clones (Scenario 1)

Social Factors

Social Activity with You
Has Collaborated with Me on Many Files
Has Submitted Many Commits with Me

Social Activity within the Organization
Has Collaborated on Many Files with Others
Has Submitted Many Commits with Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Third Developer

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons

Any Other Comments

Select Third Developer

Technical Factors
Has worked on Clones (Scenario 1)

Social Factors

Social Activity with You
Has Collaborated with Me on Many Files
Has Submitted Many Commits with Me

Social Activity within the Organization
Has Collaborated on Many Files with Others
Has Submitted Many Commits with Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Next Scenario

Figure 4.9: Scenario-2 Developer Rankings

4.3.1.3. Scenario-3

We used Scenario-3, the last one, to come up with the last ranking. We asked the judges to suppose that they had already started using our ERSF and they should make their decisions based on this. Like Scenario-2, the judges were given a list of developers but this time with characteristics representing their social relationships within the ERSF; these characteristics describe the developer trust, response, help, and the developers recommended to the Active

Developers as well as to others within the ERSF. In addition, we also displayed the data from Scenario-1 (technical expertise) and Scenario-2 (sociality within the DSSD) (Figure 4.10 and 4.11). We also highlighted the developer names who were selected in previous scenarios: yellow for Scenario-1, green for Scenario-2, and blue for both Scenario-1 and Scenario-2. Appendix A contains a full list of developers with sociality within the ERSF.

Scenario 3

We are concerned when we develop our recommender system with improving its performance and its recommendations. We do that by tracking the developers' communications through the system. We then use these communications to improve the closeness measurements of the developers to the Active Developer and improve the sociality measurements of those developers.

Scenario 3 studies the improvement of the closeness and sociality measurements. It improves the measurements of closeness by capturing:

- Trust of the Active Developer to others, which is captured when the Active Developer selects a developer to get help from, the system counts this selection as a trust of the Active Developer to the selected one.
- Response of the developers to the Active Developer's request; each time a selected developer accepts the Active Developer's request, the system counts this acceptance as a Response from the selected developer to the Active Developer.
- Help of the developers to the Active Developer. If the selected developer could help the Active Developer, the system counts this as a Help from the selected developer to the Active Developer
- Sometimes, if the selected developer could not help the Active Developer, the system allows him to recommend another developer who might be an expert and can help. Thus, the system captures this recommendation and uses it as one of the factors to measure closeness of the developers to the Active Developer

However, the system improves the measurements of a developer's sociality by capturing:

- Trust of the developers to this developer; each time any developer selects this developer, the system counts this selection as a trust to this developer.
- Response of this developer to others. When this developer accepts a request from others to help, the system counts this acceptance as a response from this developer to others.
- Help of the developer to others. This is counted each time this developer helps other developers
- Recommended of the developers by others. When any developer recommends this developer to others, the system counts this recommendation to this developer.

The following table shows the developers' closeness to you and their sociality through the recommender system. Moreover, it includes the factors of Scenario 1 and Scenario 2. The yellow highlight shows developers whom you selected in Scenario 1, the green highlight shows the developers whom you selected in Scenario 2, and the blue highlight shows the developers you have selected in both Scenario 1 and Scenario 2.

Figure 4.10: Scenario-3 Description

Developer Name		Social Activity with You					Social Activity within the Organization						
		Number of Files you Both have Cooperated on	Number of Commits you Both have Submitted	Number of Times you Have Trusted Him/Her	Number of Times He/She has Responded to Your Request	Number of Times He/She has Helped You	Number of Times He/She has been Recommended to Help You	Number of Files a Developer has Cooperated on the Modification	Number of Commits a Developer has Cooperated on the Submissions	Number of Times He/She has been Trusted by Others	Number of Times He/She has Responded to Others' Request	Number of Times He/She has Helped Others	Number of Times He/She has been Recommended to Help by Others
Photo	Ryan Boggs	0	0	0	0	0	0	0	0	26	23	18	20
		cloneTypes		Type-2	noOfFragments		2	noOfLines		19	dateOfLastModified		2012-05-04
Photo	Dguder	0	0	0	0	0	0	2	6	0	0	0	0
Photo	Rmboggs	1	2	0	0	0	0	2	4	0	0	0	0
Photo	Charles Chan	10	23	0	0	0	0	10	23	0	0	0	0
Photo	Drieseng	10	20	15	11	9	2	30	60	36	25	15	19
		cloneTypes		Type-1, Type-3	noOfFragments		3	noOfLines		17	dateOfLastModified		2004-08-11
Photo	James Geurts	4	9	0	0	0	0	10	54	0	0	0	0
Photo	Ian MacLean	2	6	0	0	0	0	2	6	0	0	0	0

Figure 4.11: Developer Social Heuristics within ERSF

Finally, we asked these judges to rank the first three developers and the reasons for their selections, which include the “Has worked on similar code fragments (Scenario-1)” option for the technical group, “Has good sociality (Scenario-2)” for sociality within the DSSD group, and the eight social heuristics of the last group (Figure 4.12).

Developer Ranking

From the previous information of the developers' closeness to you and their sociality within the organization, we would like you to rank those developers based on whom you will pick to get help from, and why you think this developer is the best one to contact and get help from.

First Developer

Select First Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons *Any Other Comments*

Technical Factors
Has worked on Clones (Scenario 1)

Social Activity
Has Good Sociality (Scenario 2)

Social Activity through Our Recommender System

Social Activity with You
Has been Trusted by me
Has Responded to my Requests
Has Helped me
Has been Recommended to Help me

Social Activity within the Organization
Has been Trusted by Others
Has Responded to Others' Requests
Has Helped Others
Has been Recommended to Help by Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Second Developer

Select Second Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons *Any Other Comments*

Technical Factors
Has worked on Clones (Scenario 1)

Social Activity
Has Good Sociality (Scenario 2)

Social Activity through Our Recommender System

Social Activity with You
Has been Trusted by me
Has Responded to my Requests
Has Helped me
Has been Recommended to Help me

Social Activity within the Organization
Has been Trusted by Others
Has Responded to Others' Requests
Has Helped Others
Has been Recommended to Help by Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Third Developer

Select Third Developer ▼

I picked up this developer because he/she

Press "Ctrl/ Command + Left-click" to select more than one reasons *Any Other Comments*

Technical Factors
Has worked on Clones (Scenario 1)

Social Activity
Has Good Sociality (Scenario 2)

Social Activity through Our Recommender System

Social Activity with You
Has been Trusted by me
Has Responded to my Requests
Has Helped me
Has been Recommended to Help me

Social Activity within the Organization
Has been Trusted by Others
Has Responded to Others' Requests
Has Helped Others
Has been Recommended to Help by Others

Any Other Comments
Others

If you Select "Others", Please Enter your Comments here

Next Scenario

Figure 4.12: Scenario-3 Developer Rankings

4.3.2. Determining Heuristic and Group Weights Phase

The results from the Collecting Human Rankings phase include three parts: the ranked developers, the considered heuristics, and the given rankings to these developers by the judges. In this phase, we only used both the considered heuristics and the given rankings to be analyzed in order to determine the importance of the heuristics within their groups as well as the groups among each other.

We used the Weka tool, which is a collection of machine learning algorithms for data mining tasks. The algorithms are applied to a dataset to learn from, analyze its structural patterns, and make some predictions [29]. Weka has many techniques, such as data pre-processing, classification, regression, clustering, association rules, selecting attributes, and visualization.

In this phase, we used the selection attributes technique, which analyzes the dataset and predicts values of attributes within that set. We used this technique in order to analyze the heuristics that the human rankers considered and assign them a weight reflecting their degree of importance. We applied the Filtered Attribute Evaluation method, which is a specific attribute selection technique, to each of the scenarios to find the weights of the heuristics under them. In other words, we applied the Filtered Attribute Evaluation method to the selected technical heuristics (reasons) of Scenario-1, to the selected social heuristics (reasons) of Scenario-2, and to the selected social heuristics (reasons) of Scenario-3 to find out their weights. Moreover, we applied this method to the extracted groups (technical group, social group within the DSSD, and social group within the ERSF) from Scenario-3 since it combined all of the 16 heuristics, to determine the weight and importance of each group.

Our analysis in this phase was done on the rankings of each scenario as follow:

1. For each judge and for each of his/her rankings, we extracted the heuristics s/he considered while ranking a particular developer, and we replaced them with their values that represent the ranked developer expertise/sociality, depends on the scenario being working on.
2. We also extracted the ranking the judge gave to the developer in order to analyze the importance of these heuristics from the judge perspective.
3. Above data (steps1 and 2) are then used to create an instance representing the judge decisions to be used as an input to the Weka tool.
4. Finally, after all the instances were created, we applied the Filtered Attribute Evaluation method on these instances. Weka then analyzed them in order to determine the weight of each heuristic as to its level of importance among each other.

Below we present our analysis results for each scenario/heuristics group and the groups themselves.

4.3.2.1. Technical Heuristics Weights

In this section, we are concerned with determining the weights of each technical heuristic (types: Type-1, Type-2, Type-3; number of fragments, number of lines, and most recent modifications). Table 4.1 presents the judges' rankings and the technical heuristics they considered while they were ranking the developers in Scenario-1.

Table 4.1: Scenario-1 Judge Rankings

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-1	1	Ryan Boggs	Number of fragments Most recent modification
	2	Drieseng	Number of lines Most recent modification
	3	Scott Hernandez	Type-1 Type-2
Judge-2	1	Gerry Shaw	Type-3
	2	Ryan Boggs	Most recent modification
	3	Scott Hernandez	Type-1 Type-2
Judge-3	1	Michael C. Two	Number of lines
	2	Gerry Shaw	Number of lines
	3	Ryan Boggs	Number of lines
Judge-4	1	Ryan Boggs	Type-2 Most recent modification
	2	Scott Hernandez	Type-2
	3	Michael C. Two	Number of lines
Judge-5	1	Scott Hernandez	Most recent modification
	2	Drieseng	Number of fragments
	3	Michael C. Two	Number of lines
Judge-6	1	Scott Hernandez	Type-1 Type-2 Number of lines
	2	Ryan Boggs	Type-2 Number of fragments
	3	Gerry Shaw	Type-3 Number of lines Most recent modification
Judge-7	1	Ryan Boggs	Number of lines Most recent modification
	2	Drieseng	Type-1 Number of fragments Number of lines
	3	Scott Hernandez	Number of fragments Number of lines
Judge-8	1	Ryan Boggs	Most recent modification

Judges	Rankings	Developers	Considered Technical Heuristics
	2	Gerry Shaw	Type-3
	3	Drieseng	Number of fragments
Judge-9	1	Scott Hernandez	Type-2 Type-3 Number of fragments
	2	Drieseng	Type-1
	3	Ryan Boggs	Type-2
Judge-10	1	Drieseng	Type-1 Type-3 Number of fragments
	2	Scott Hernandez	Type-1 Type-2
	3	Ryan Boggs	Number of fragments

This shows how an instance is created following step 1, 2, and 3. For instance, Judge-6 ranked “Gerry Shaw” with “3” because he has worked on “Type-3”, modified “Number of lines”, and done “Most recent modifications”. Accordingly, we replaced the “Type-3” heuristic with “1” (The Type’s value is either “0” or “1”), “Number of lines” with “12”, “Most recent modifications” with the number of days since the date of the last modifications (2002-08-14), and filled other heuristics with “0” in order to create an instance as an input for Weka as shown in Table 4.2.

Table 4.2: Technical Heuristics (the Filtered Attribute Evaluation Method Input Example)

Values	Technical Expertise
<instance>	
<value>0</value>	Type-1
<value>0</value>	Type-2
<value>1</value>	Type-3
<value>0</value>	Number of fragments
<value>12</value>	Number of lines
<value>3955</value>	Most recent modification
<value>3</value>	Ranking
</instance>	

Figure 4.13 shows the resulted technical heuristic weights from step 4, which is concerned with applying the Filtered Attribute Evaluation method in Weka.

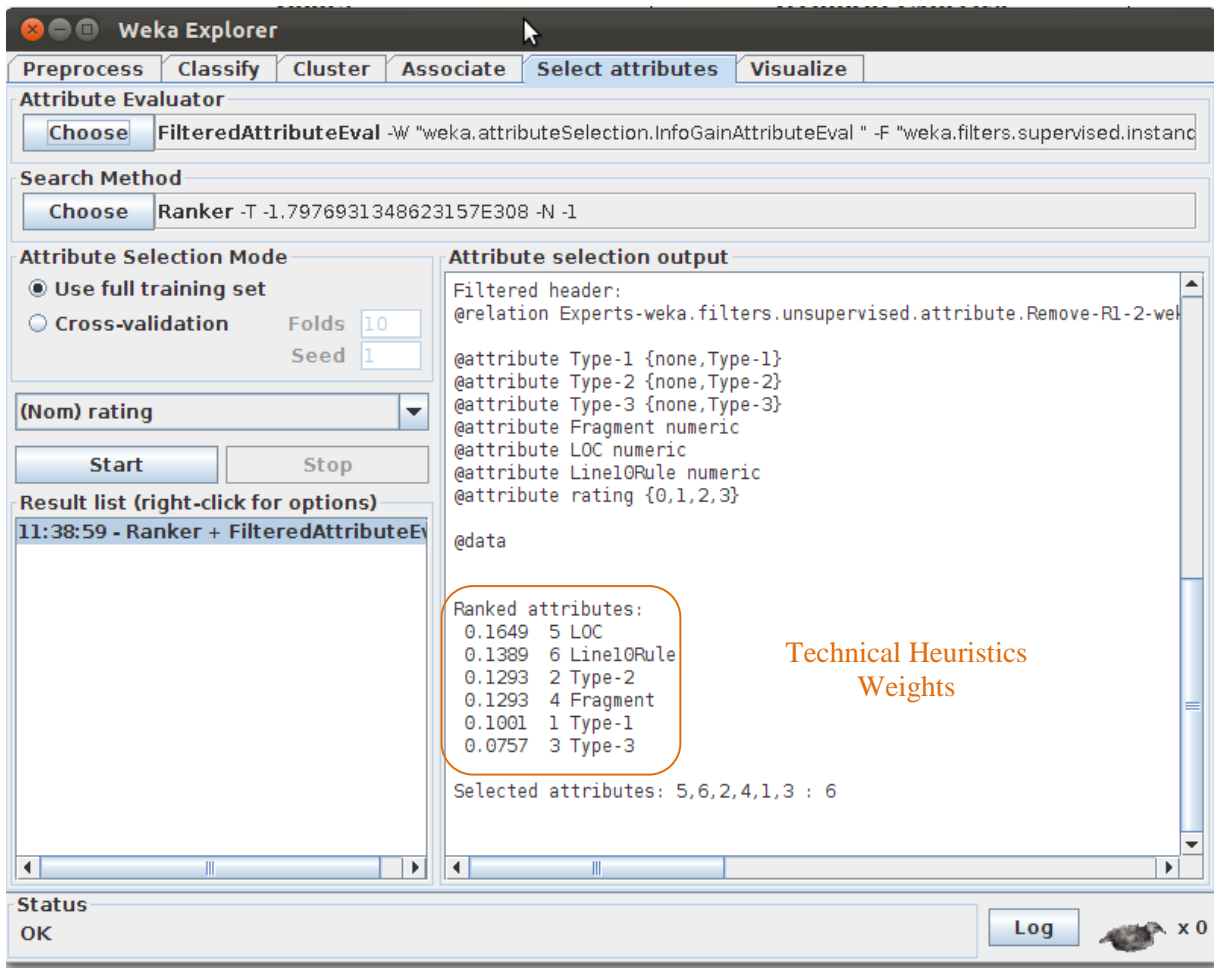


Figure 4.13: Technical Heuristics Weights by Weka

4.3.2.2. Social Heuristics within DSSD Weights

This section is concerned with determining the weights of each social heuristic within the DSSD (number of shared files and commits heuristics with the Active Developer as well as the number of shared files and commits heuristics within the DSSD). Table 4.3 presents the judges' rankings and the social heuristics within the DSSD they considered while they were ranking the developers in Scenario-2.

Table 4.3: Scenario-2 Judge Rankings

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-1	1	Drieseng	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared cmmits Sociality within the DSSD: Number of shared files Number of shared commits
	2	Scott Hernandez	Has worked on Clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
	3	Charles Chan	Sociality with the Active Developer: Number of shared files Number of shared commits
Judge-2	1	Jarek Kowalski	Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
	2	Gerry Shaw	Has worked on clones (Scenario 1) Sociality within the DSSD: Number of shared files Number of shared commits
	3	claytonharbour	Sociality within the DSSD: Number of shared files Number of shared commits
Judge-3	1	Michael C. Two	Has worked on clones (Scenario 1)
	2	Gerry Shaw	Has worked on clones (Scenario 1) Sociality within the DSSD: Number of shared files Number of shared commits
	3	Scott Hernandez	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits
Judge-4	1	Drieseng	Sociality within the DSSD: Number of shared files
	2	Charles Chan	Sociality with the Active Developer:

Judges	Rankings	Developers	Considered Technical Heuristics
			Number of shared commits
	3	Dguder	Sociality within the DSSD: Number of shared files Number of shared commits
Judge-5	1	Scott Hernandez	Sociality with the Active Developer: Number of shared files Number of shared commits
	2	Charles Chan	Sociality with the Active Developer: Number of shared files Number of shared commits
	3	Drieseng	Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
Judge-6	1	Scott Hernandez	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
	2	Gerry Shaw	Has worked on clones (Scenario 1) Sociality within the DSSD: Number of shared files Number of shared commits
	3	Charles Chan	Sociality with the Active Developer: Number of shared files Number of shared commits
Judge-7	1	Drieseng	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
	2	Scott Hernandez	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits

Judges	Rankings	Developers	Considered Technical Heuristics
	3	Gerry Shaw	Has worked on clones (Scenario 1) Sociality within the DSSD: Number of shared files Number of shared commits
Judge-8	1	Drieseng	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits
	2	Gerry Shaw	Has worked on clones (Scenario 1)
	3	Ryan Boggs	Has worked on clones (Scenario 1)
Judge-9	1	Drieseng	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits
	2	Scott Hernandez	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits
	3	Ryan Boggs	Has worked on clones (Scenario 1)
Judge-10	1	Drieseng	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files
	2	Scott Hernandez	Has worked on clones (Scenario 1) Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits
	3	Charles Chan	Sociality with the Active Developer: Number of shared files Number of shared commits Sociality within the DSSD: Number of shared files Number of shared commits

This shows how an instance is created following step 1, 2, and 3. However, in this part we ignored the judge consideration of the “Clones” since we were just concerned with determining

the weights of the social heuristics within the DSSD in terms of their relative importance among each other. For example, Judge-5 ranked “Drieseng” with a “3” because he had “number of shared files” and “number of shared commits” with him/her as well as “number of shared files” and “number of shared commits” with other developers within the DSSD. Accordingly, we replaced the “number of shared files” and “number of shared commits” heuristics with the Active Developer with “10” and “20”, as well as replacing the “number of shared files” and the “number of shared commits” within the DSSD with “30” and “60” in order to create an instance as an input for Weka as shown in Table 4.4.

Table 4.4: Social Heuristics within the DSSD (the Filtered Attribute Evaluation Method Input Example)

Values	Social Heuristics within the DSSD
<instance>	
<value>10</value>	Number of shared files with the Active Developer
<value>20</value>	Number of shared commits with the Active Developer
<value>30</value>	Number of shared files within the DSSD
<value>60</value>	Number of shared commits within the DSSD
<value>3</value>	Ranking
</instance>	

Figure 4.14 shows the resulting social heuristic within the DSSD weights from step 4, which is concerned with applying the Filtered Attribute Evaluation method in Weka.

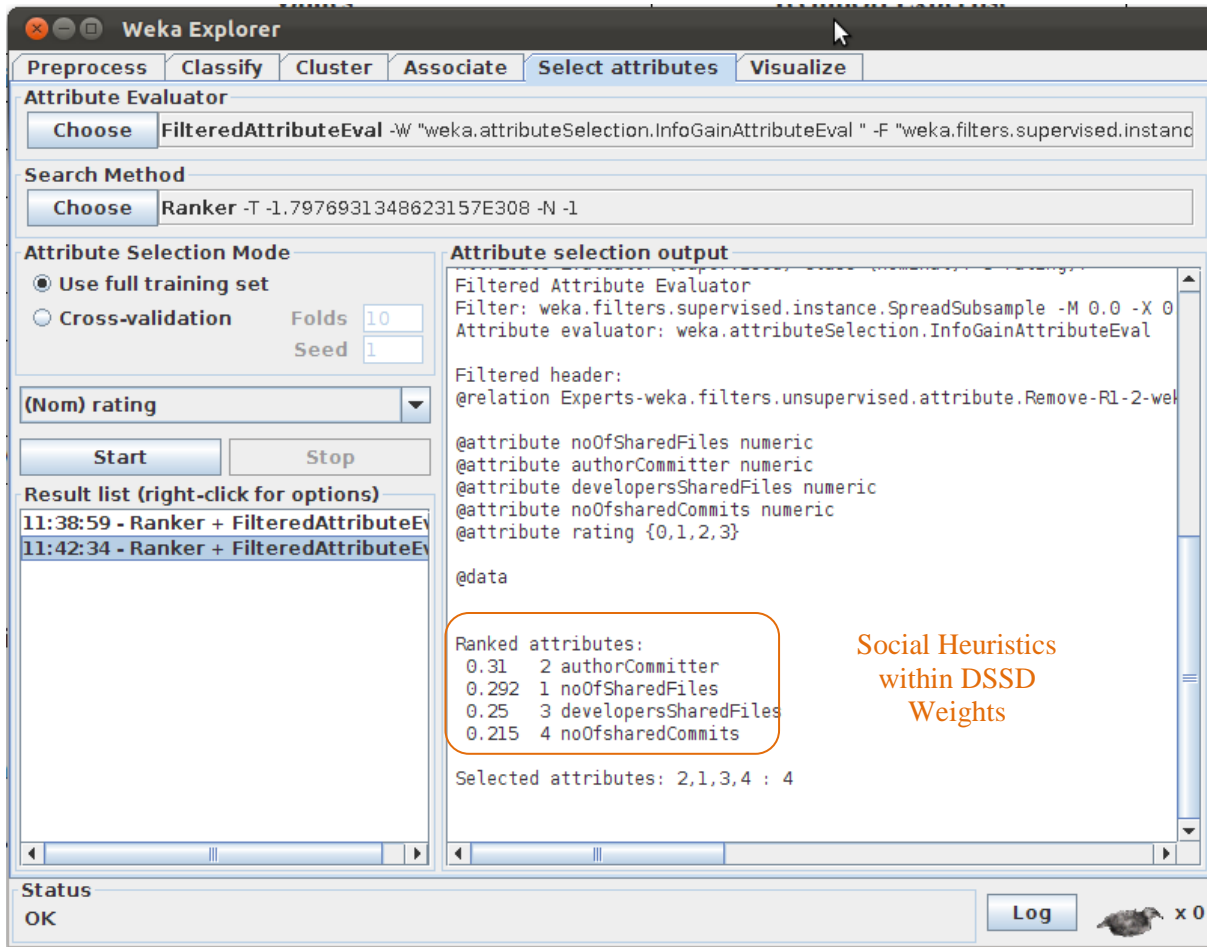


Figure 4.14: Social Heuristics within DSSD Weights by Weka

4.3.2.3. Social Heuristics within ERSF Weights

This section is concerned with determining the weights of each social heuristic within the ERSF (trust, response, help, and recommended to the Active Developer as well as trust, response, helpfulness, and recommended within the ERSF). Table 4.5 presents the judges' rankings and the social heuristics within the ERSF they considered while they were ranking the developers in Scenario-3.

Table 4.5: Scenario-3 Judge Rankings

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-1	1	Drieseng	Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	2	Michael C. Two	Sociality within the ERSF: Trust Response Help Recommended
	3	Bernard Vander Beken	Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
Judge-2	1	Drieseng	Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended

Judges	Rankings	Developers	Considered Technical Heuristics
	2	Jarek Kowalski	Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	3	Michael C. Two	Sociality within the ERSF: Trust Response Help Recommended
Judge-3	1	Michael C. Two	Has worked on Clones (Scenario 1) Sociality with the Active Developer: Trust Response Help Recommended
	2	Gerry Shaw	Has worked on clones (Scenario 1)
	3	Ryan Boggs	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response Help Recommended
Judge-4	1	Jarek Kowalski	Sociality with the Active Developer: Trust Response Help Recommended
	2	Michael C. Two	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended

Judges	Rankings	Developers	Considered Technical Heuristics
	3	Ryan Boggs	Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
Judge-5	1	Drieseng	Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	2	Jarek Kowalski	Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	3	Bernard Vander Beken	Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
Judge-6	1	Ryan Boggs	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response Help Recommended

Judges	Rankings	Developers	Considered Technical Heuristics
	2	Michael C. Two	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response Help Recommended
	3	Drieseng	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response Help Recommended
Judge-7	1	Drieseng	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	2	Michael C. Two	Has worked on clones (Scenario 1) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	3	Ryan Boggs	Has worked on clones (Scenario 1) Sociality with the Active Developer: Trust Response Help Recommended

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-8	1	Drieseng	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help
	2	Michael C. Two	Has worked on clones (Scenario 1) Sociality with the Active Developer: Trust Response Help
	3	Ryan Boggs	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended
Judge-9	1	Drieseng	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Recommended Sociality within the ERSF: Trust Response Help Recommended
	2	Ryan Boggs	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response Help Recommended
	3	Scott Hernandez	Has worked on clones (Scenario 1)

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-10	1	Drieseng	Has worked on clones (Scenario 1) Has good sociality (Scenario 2) Sociality with the Active Developer: Trust Response Help Sociality within the ERSF: Trust Response Help
	2	Ryan Boggs	Has worked on clones (Scenario 1) Sociality within the ERSF: Trust Response
	3	Jarek Kowalski	Sociality with the Active Developer: Trust Response Help Sociality within the ERSF: Trust Response Help

This shows how an instance is created following steps 1, 2, and 3. However, in this part we ignored the judges’ consideration of the “Clones” and “Sociality within the DSSD” since we were just concerned with determining the weights of the social heuristics within the ERSF as the relative importance among each other. For example, Judge-7 ranked “Drieseng” with a “1” because he has “trust”, “response”, “help”, “recommended” to him/her as well as has “trust”, “response”, “help”, “recommended” to other developers within the ERSF. Accordingly, we replaced the “trust”, “response”, “help”, “recommended” heuristics with the Active Developer with “15”, “11”, “9”, and “2”, as well as replacing the “trust”, “response”, “help”, “recommended” within the ERSF with “36”, “25”, “15”, and “19” in order to create an instance as input for Weka as shown in Table 4.6.

Table 4.6: Social Heuristics within the ERSF (the Filtered Attribute Evaluation Method Input Example)

Values	Social Heuristics within the ERSF
<instance>	
<value>10</value>	Trust by the Active Developer
<value>20</value>	Response to the Active Developer
<value>30</value>	Help to the Active Developer
<value>60</value>	Recommended to the Active Developer
<value>10</value>	Trust within the ERSF
<value>20</value>	Response within the ERSF
<value>30</value>	Helpfulness within the ERSF
<value>60</value>	Recommended within the ERSF
<value>1</value>	Ranking
</instance>	

Figure 4.15 shows the resulting social heuristic within the ERSF weights from step 4, which is concerned with applying the Filtered Attribute Evaluation method in Weka.

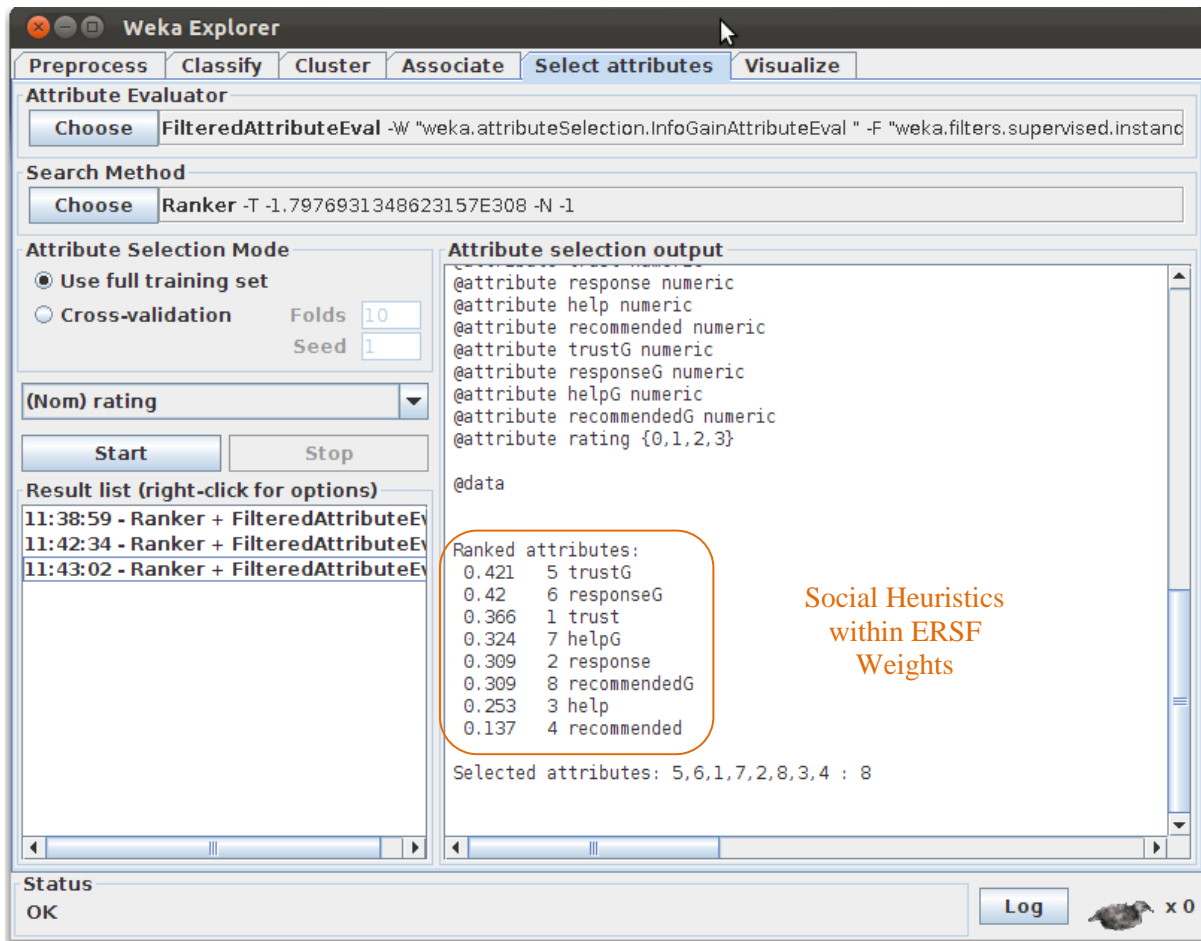


Figure 4.15: Social Heuristics within ERSF Weights by Weka

4.3.2.4. Groups Weights

In the previous classifications, we were concerned with determining the weights of each heuristic within its group. However, in this classification, we were concerned with determining the group weights relative to each other. Thus, we analyzed the data in Table 4.5 by replacing the “Has worked on similar code fragments (Scenario-1)” with “Technical expertise” and “Has good sociality (Scenario-2)” with “Sociality within the DSSD”, and we replaced the social heuristics within the ERSF with its group “Sociality within the ERSF”. Table 4.7 represents the three groups.

Table 4.7: Scenario-3 Groups Judge Rankings

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-1	1	Drieseng	Sociality within the DSSD Sociality within the ERSF
	2	Michael C. Two	Sociality within the ERSF
	3	Bernard Vander Beken	Sociality within the ERSF
Judge-2	1	Drieseng	Sociality within the DSSD Sociality within the ERSF
	2	Jarek Kowalski	Sociality within the DSSD Sociality within the ERSF
	3	Michael C. Two	Sociality within the ERSF
Judge-3	1	Michael C. Two	Technical Expertise Sociality within the ERSF
	2	Gerry Shaw	Technical Expertise
	3	Ryan Boggs	Technical Expertise Sociality within the ERSF
Judge-4	1	Jarek Kowalski	Sociality within the ERSF
	2	Michael C. Two	Technical Expertise Sociality within the DSSD Sociality within the ERSF
	3	Ryan Boggs	Sociality within the ERSF
Judge-5	1	Drieseng	Sociality within the ERSF
	2	Jarek Kowalski	Sociality within the ERSF
	3	Bernard Vander Beken	Sociality within the ERSF
Judge-6	1	Ryan Boggs	Technical Expertise Sociality within the ERSF
	2	Michael C. Two	Technical Expertise Sociality within the ERSF
	3	Drieseng	Technical Expertise Sociality within the ERSF
Judge-7	1	Drieseng	Technical Expertise Sociality within the DSSD Sociality within the ERSF
	2	Michael C. Two	Technical Expertise Sociality within the ERSF
	3	Ryan Boggs	Technical Expertise Sociality within the ERSF

Judges	Rankings	Developers	Considered Technical Heuristics
Judge-8	1	Drieseng	Technical Expertise Sociality within the DSSD Sociality within the ERSF
	2	Michael C. Two	Technical Expertise Sociality within the ERSF
	3	Ryan Boggs	Technical Expertise Sociality within the DSSD Sociality within the ERSF
Judge-9	1	Drieseng	Technical Expertise Sociality within the DSSD Sociality within the ERSF
	2	Ryan Boggs	Technical Expertise Sociality within the ERSF
	3	Scott Hernandez	Technical Expertise
Judge-10	1	Drieseng	Technical Expertise Sociality within the DSSD Sociality within the ERSF
	2	Ryan Boggs	Technical Expertise Sociality within the ERSF
	3	Jarek Kowalski	Sociality within the ERSF

This shows how an instance is created following steps 1, 2, and 3. However, in step 1, we replaced the heuristic groups they considered with a “1” (The values of these groups were either “0” or “1”). For example, Judge-6 ranked “Ryan Boggs” with a “1” because he had “Technical Expertise”, and he had “Sociality within the ERSF”. Accordingly, we replaced these two groups with a “1” and filled the “Sociality within the DSSD” with a “0” in order to create an instance as input for Weka as shown in Table 4.8.

Table 4.8: Heuristic Groups (the Filtered Attribute Evaluation Method Input Example)

Values	Represented Data
<instance>	
<value>1</value>	Technical Expertise
<value>0</value>	Sociality within the DSSD
<value>1</value>	Sociality within the ERSF
<value>1</value>	Ranking
</instance>	

Figure 4.16 shows the resulting group weights from step 4, which is concerned with applying the Filtered Attribute Evaluation method in Weka.

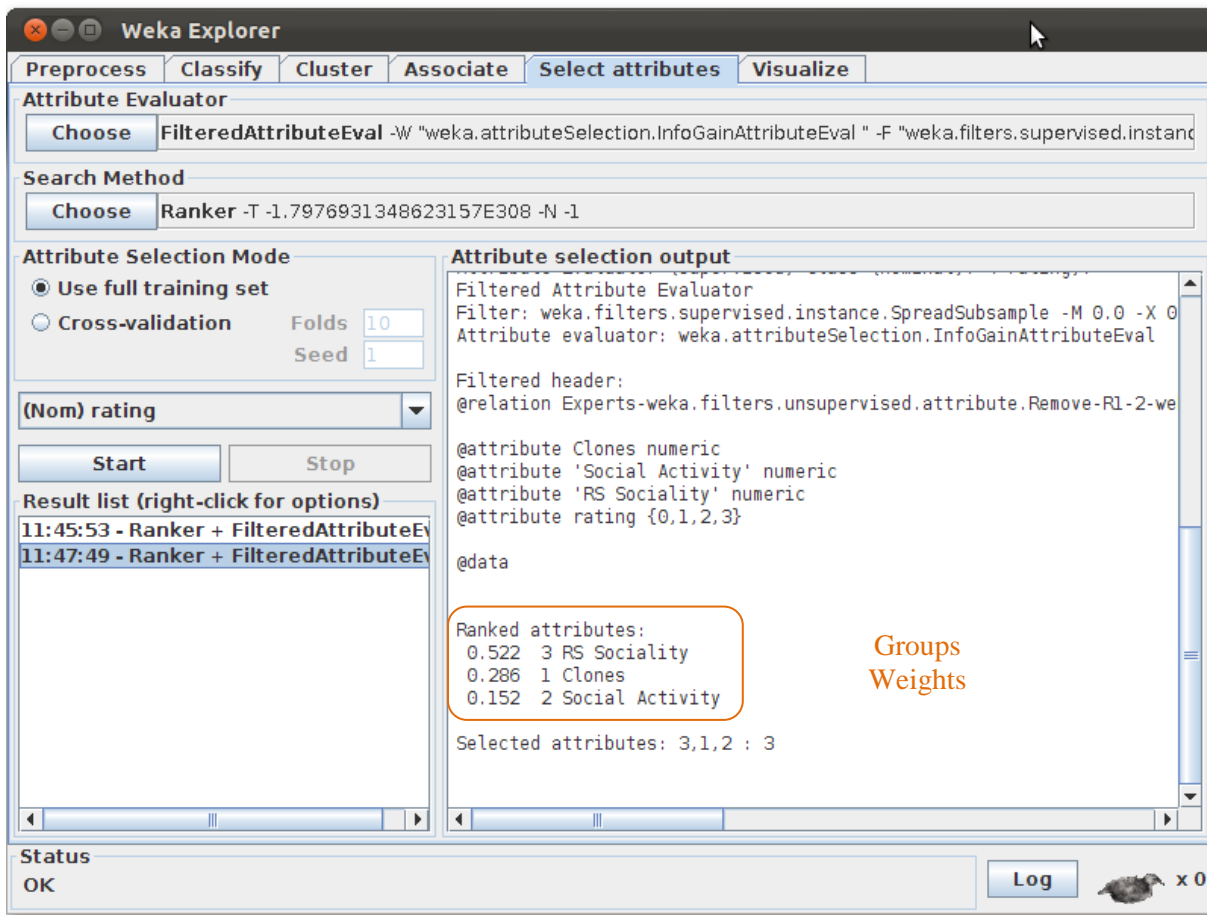


Figure 4.16: Heuristic Group Weights by Weka

4.3.3. Designing Recommender System Algorithm Phase

In this section, we explain how we used the weights that resulted from the Weka analysis to prioritize the heuristics and groups in computing the developer likelihoods to be experts and help the Active Developer using our algorithm (Formula-1):

$$D_e = \sum_{h=1}^n gw \left(\frac{D_{(s/e)}}{T_{(s/e)}} \times hw \right) \quad 0 \leq D_e \leq 1 \quad \text{Formula-1}$$

Where D_e is the current developer for whom we are computing his/her likelihood to be an expert based on his/her technical heuristics and/or sociality, h is the current heuristic the ratio is computed under, n is the total number of heuristics, gw is the group weight where this heuristic is classified under (technical heuristic, social heuristic within DSSD, or social heuristic within ERSF), $D(e/s)$ is the current developer expertise/sociality under this heuristic, $T(e/s)$ is the total expertise/sociality under this heuristic, and hw is the heuristic weight.

For instance, we have two developers (“Dguder” and “Dmitry Jemerov”). “Dguder” has “2 out of 151 shared files” and “6 out of 425 shared commits” within the DSSD; on the other hand, “Dmitry Jemerov” has “21 out of 164 trust”, “18 out of 128 response”, “14 out of 84 helpfulness”, and “21 out of 126 recommended” within ERSF. From Weka analysis, we came up with the weights in Table 4.9.

Table 4.9: Heuristic and Group Weights by Weka Example

Groups	Heuristics	Heuristic Weights (<i>hw</i>)	Group Weights (<i>gw</i>)
Sociality within the DSSD	Number of Shared Files	0.25	0.152
	Number of Shared Commits	0.215	
Sociality within the ERSF	Trust	0.421	0.522
	Response	0.42	
	Helpfulness	0.324	
	Recommended	0.309	

We thus applied our algorithm to each of the two developer characteristics to compute their likelihood to be an expert as follows:

$$Dgudere = [0.152 ((2/151) * 0.25 + (6/425) * 0.215)] = 0.001$$

$$Dmitry Jemerove = [0.522 ((21/164) * 0.421 + (18/128) * 0.42 + (14/84) * 0.324 + (21/126) * 0.309)] = 0.11$$

The calculation shows that “Dmitry Jemerov” has a higher result than “Dguder”. Thus, “Dmitry Jemerov” has more likelihood to be an expert to help the Active Developer than “Dguder” and should be ranked as the first developer to contact.

4.3.4. Evaluating the Accuracy of the Algorithm Phase

One other purpose for our experiment is evaluating the accuracy of our algorithm in recommending suitable experts to assist the Active Developers completing the code at hand. We did this for each scenario by comparing our algorithm rankings to the judges’ rankings as well as to NaiveBayes, NaiveNet, and J48 classifiers in Weka.

The three classifiers apply their algorithms on a real dataset to model numeric attributes in order to make decisions that are independent of each other and equally important [29]. We used the three classifiers in our experiment since we needed algorithms that predict rankings based on independent numeric attributes. In other words, we used NaiveBayes NaiveNet, and J48 to learn

from the judges' rankings and the considered heuristics, which are numerically independent heuristics, and analyze its structural patterns and the developer rankings (i.e. 1, 2, or 3), with which we then compared our algorithm rankings in order to evaluate its accuracy.

Our evaluation in this phase was done as follows:

1. We restructured for each judge and for each of his/her rankings the ranked developer, the considered heuristics, and the given ranking to this developer by the judge from the results of the Human Rankings Collection phase.
2. This data is then used to design the input for both our algorithm as well as the NaiveBayes, NaïveNet, and J48 machine learning algorithms. Thus, for each judge and each of his/her rankings, we created an instance. The instance includes the judge's ID, ranked developer's ID, and the considered heuristics values that represent the developer's expertise/sociality, depending on the scenario being working on. However, the difference between the input instance to our algorithm and machine learning algorithms is that our algorithm does not need the developer ranking to predict his/her ranking. On the other hand, the machine learning algorithms need the judge's ranking from which to learn and predict the developer's ranking based on the entered heuristics. Since we have 10 judges and each of them has ranked the top three experts in this scenario, we have 30 instances in total.
3. After we created all the judge instances, we applied our algorithm to the heuristic values in these instances (as we explained in section 4.3.3) in order to find the algorithm ranking to the developers in the instances. Then, we compared for each

instance the judge ranking and our algorithm ranking in order to evaluate its performance.

4. We also applied the NaiveBayes, NaiveNet, and J48 machine learning algorithms to the heuristic values in the instances in order to learn from and predict the developer rankings in the instances. Then, we compared for each instance our algorithm's ranking to the predicted rankings by each of the machine learning algorithms in order to evaluate its performance.

In the following section, we show the results from our analysis for each scenario and the group as well.

Regarding the machine learning algorithms, we will limit our explanation in this section to the rankings by the NaiveBayes. Then, in the Discussion (section 4.5.1), we will explain about the NaiveNet and J48 machine learning algorithms.

4.3.4.1. Scenario-1 Rankings Comparison

Scenario-1 was concerned with ranking the top three developers based on their technical expertise on the code fragments that are similar to the one the Active Developer needs help with. Also, it was concerned with capturing the technical heuristics that judges considered while they were ranking the developers.

This shows an example of creating an instance following step 1 and 2. For example, Table 4.1 shows that Judge-6 ranked "Gerry Shaw" with a "3" because he worked on "Type-3" clone, modified "Number of lines" and did the "Most recent modification". Accordingly, we replaced the Judge-6 with "J6"; "Drieseng" with "D5"; and the "Type-3" heuristic with "1", "Number of lines" with "12", "Most recent modification" with the number of days since the date of the last

modification (2002-08-14), and filled other heuristics with “0” in order to create the instance of Judge-6 and his/her third ranking as shown in Table 4.10.

Table 4.10: Scenario-1 Rankings (RS Algorithm and NaiveBayes Input Example)

Values	Technical Expertise
<instance ID = "1">	
<value>J6</value>	Judge's ID
<value>D5</value>	Ranked Developer's ID
<value>0</value>	Type-1
<value>0</value>	Type-2
<value>1</value>	Type-3
<value>0</value>	Number of fragments
<value>12</value>	Number of lines
<value>3955</value>	Most recent modification
<value>3</value>	Ranking
</instance>	

Table 4.11 presents the three rankings: Judge rankings, NaiveBayes rankings, and our algorithm rankings from applying step 3 and 4.

Table 4.11: Scenario-1 Judge, NaiveBayes, and RS Algorithm Rankings Comparisons

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
1	Judge-1	Ryan Boggs	1	1	2
2		Drieseng	2	2	3
3		Scott Hernandez	3	1	1
4	Judge-2	Gerry Shaw	1	2	2
5		Ryan Boggs	2	1	3
6		Scott Hernandez	3	3	1
7	Judge-3	Michael C. Two	1	3	3
8		Gerry Shaw	2	2	2
9		Ryan Boggs	3	3	1

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
10	Judge-4	Ryan Boggs	1	1	1
11		Scott Hernandez	2	3	2
12		Michael C. Two	3	3	3
13	Judge-5	Scott Hernandez	1	1	3
14		Drieseng	2	2	1
15		Michael C. Two	3	3	2
16	Judge-6	Scott Hernandez	1	3	1
17		Ryan Boggs	2	1	2
18		Gerry Shaw	3	2	3
19	Judge-7	Ryan Boggs	1	1	3
20		Drieseng	2	2	1
21		Scott Hernandez	3	3	2
22	Judge-8	Ryan Boggs	1	1	3
23		Gerry Shaw	2	2	2
24		Drieseng	3	2	1
25	Judge-9	Scott Hernandez	1	3	1
26		Drieseng	2	2	3
27		Ryan Boggs	3	1	2
28	Judge-10	Drieseng	1	2	1
29		Scott Hernandez	2	3	2
30		Ryan Boggs	3	1	3

4.3.4.2. Scenario-2 Rankings Comparison

Scenario-2 was concerned with ranking the top three developers based on their social relationships within the DSSD. Also, it was concerned with capturing the social heuristics that judges considered while they were ranking the developers.

This shows an example of creating an instance following steps 1 and 2. For example, Table 4.3 shows that Judge-5 ranked “Drieseng” with a “3” because he had “number of shared files” and “number of shared commits” with him as well as “number of shared files” and “number of

shared commits” with other developers within the DSSD. Accordingly, we replaced Judge-5 with “J5”; “Drieseng” with “D16”; and the “number of shared files” and “number of shared commits” heuristics with the Active Developer with “10” and “20”, as well as replacing the “number of shared files” and the “number of shared commits” within the DSSD with “30” and “60” in order to create an instance as input for Weka as shown in Table 4.12.

Table 4.12: Scenario-2 Rankings (RS Algorithm and NaiveBayes Input Example)

Values	Social Heuristics within the DSSD
<instance>	
<value>J5</value>	Judge’ ID
<value>D16</value>	Ranked Developer’s ID
<value>10</value>	Number of shared files with the Active Developer
<value>20</value>	Number of shared commits with the Active Developer
<value>30</value>	Number of shared files within the DSSD
<value>60</value>	Number of shared commits within the DSSD
<value>3</value>	Ranking
</instance>	

Table 4.13 presents the three rankings: Judge rankings, NaiveBayes rankings, and our algorithm rankings from applying step 3 and 4.

Table 4.13: Scenario-2 Judges, NaiveBayes, and RS Algorithm Rankings Comparisons

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
1	Judge-1	Drieseng	1	1	1
2		Scott Hernandez	2	2	3
3		Charles Chan	3	3	2
4	Judge-2	Jarek Kowalski	1	1	1

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
5		Gerry Shaw	2	2	2
6		claytonharbour	3	3	3
7	Judge-3	Michael C. Two	1	1	3
8		Gerry Shaw	2	2	2
9		Scott Hernandez	3	2	1
10	Judge-4	Drieseng	1	1	1
11		Charles Chan	2	3	2
12		Dguder	3	3	3
13	Judge-5	Scott Hernandez	1	2	3
14		Charles Chan	2	3	2
15		Drieseng	3	1	1
16	Judge-6	Scott Hernandez	1	2	2
17		Gerry Shaw	2	2	3
18		Charles Chan	3	3	1
19	Judge-7	Drieseng	1	1	1
20		Scott Hernandez	2	2	2
21		Gerry Shaw	3	2	3
22	Judge-8	Drieseng	1	1	1
23		Gerry Shaw	2	2	2
24		Ryan Boggs	3	3	3
25	Judge-9	Drieseng	1	1	1
26		Scott Hernandez	2	2	2
27		Ryan Boggs	3	3	3
28	Judge-10	Drieseng	1	1	1
29		Scott Hernandez	2	2	3
30		Charles Chan	3	3	2

4.3.4.3. Scenario-3 Rankings Comparison

Scenario-3 was concerned with ranking the top three developers based on their social relationships within the ERSF. Also, it was concerned with capturing the social heuristics that judges considered while they were ranking the developers.

This shows an example of creating an instance following step 1 and 2. For example, Judge-7 ranked “Drieseng” with a “1” because he had “trust”, “response”, “help”, “recommended” to him/her as well as has “trust”, “response”, “help”, “recommended” to other developers within the ERSF. Accordingly, we replaced Judge-7 with “J7”; “Drieseng” with “D16”; and the “trust”, “response”, “help”, “recommended” heuristics with the Active Developer with “15”, “11”, “9”, and “2”, as well as replacing the “trust”, “response”, “help”, “recommended” within the ERSF with “36”, “25”, “15”, and “19” in order to create an instance as input for Weka as shown in Table 4.14.

Table 4.14: Scenario-3 Rankings (RS Algorithm and NaiveBayes Input Example)

Values	Social Heuristics within the ERSF
<instance>	
<value>J7</value>	Judge’s ID
<value>D16</value>	Ranked Developer’s ID
<value>10</value>	Trust by the Active Developer
<value>20</value>	Response to the Active Developer
<value>30</value>	Help to the Active Developer
<value>60</value>	Recommended to the Active Developer
<value>10</value>	Trust within the ERSF
<value>20</value>	Response within the ERSF
<value>30</value>	Helpfulness within the ERSF
<value>60</value>	Recommended within the ERSF
<value>1</value>	Ranking
</instance>	

Table 4.15 presents the three rankings: Judge rankings, NaiveBayes rankings, and our algorithm rankings from applying step 3 and 4.

Table 4.15: Scenario-3 Judges, NaiveBayes, and RS Algorithm Rankings Comparisons

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
1	Judge-1	Drieseng	1	1	1
2		Michael C. Two	2	2	2
3		Bernard Vander Beken	3	3	3
4	Judge-2	Drieseng	1	1	1
5		Jarek Kowalski	2	2	2
6		Michael C. Two	3	2	3
7	Judge-3	Michael C. Two	1	2	1
8		Gerry Shaw	2	1	2
9		Ryan Boggs	3	3	3
10	Judge-4	Jarek Kowalski	1	2	1
11		Michael C. Two	2	2	2
12		Ryan Boggs	3	3	3
13	Judge-5	Drieseng	1	1	1
14		Jarek Kowalski	2	2	2
15		Bernard Vander Beken	3	3	3
16	Judge-6	Ryan Boggs	1	3	1
17		Michael C. Two	2	2	2
18		Drieseng	3	1	3
19	Judge-7	Drieseng	1	1	1
20		Michael C. Two	2	2	2
21		Ryan Boggs	3	3	3
22	Judge-8	Drieseng	1	1	1
23		Michael C. Two	2	2	2
24		Ryan Boggs	3	3	3
25	Judge-9	Drieseng	1	1	1
26		Ryan Boggs	2	3	2
27		Scott Hernandez	3	3	3
28	Judge-10	Drieseng	1	1	1
29		Ryan Boggs	2	3	2
30		Jarek Kowalski	3	2	3

4.3.4.4. Scenario-3 Group Rankings Comparison

Scenario-3 was concerned with ranking the top three developers based on their social relationships within the ERSF. Also, it was concerned with capturing the social heuristics that judges considered while they were ranking the developers. However, we also used these rankings to analyze the heuristic groups as shown in Table 4.7.

This shows an example of creating an instance following step 1 and 2. For example, Judge-7 ranked “Drieseng” with a “1” because he had “Technical expertise”, “Sociality within the DSSD”, and “Sociality within the ERSF”. Accordingly, we replaced Judge-7 with “J7”; “Drieseng” with “D16”; and the “Technical expertise”, “Sociality within the DSSD”, and “Sociality within the ERSF” groups with a “1” in order to create an instance as input for Weka as shown in Table 4.16.

Table 4.16: Scenario-3 Groups Rankings (RS Algorithm and NaiveBayes Input Example)

Values	Social Heuristics within the ERSF
<instance>	
<value>J7</value>	Judge’s ID
<value>D16</value>	Ranked Developer’s ID
<value>1</value>	Technical Expertise
<value>1</value>	Sociality within the DSSD
<value>1</value>	Sociality within the ERSF
<value>1</value>	Ranking
</instance>	

Table 4.17 presents the three rankings: Judge rankings, NaiveBayes rankings, and our algorithm rankings from applying step 3 and 4.

Table 4.17: Scenario-3 Judges, NaiveBayes, and RS Algorithm Groups Rankings Comparisons

Instances	Judges	Ranked Developers	Judge Rankings	NaiveBayes Rankings	Recommender System Algorithm Rankings
1	Judge-1	Drieseng	1	1	1
2		Michael C. Two	2	2	2
3		Bernard Vander Beken	3	3	3
4	Judge-2	Drieseng	1	1	1
5		Jarek Kowalski	2	2	2
6		Michael C. Two	3	2	3
7	Judge-3	Michael C. Two	1	2	1
8		Gerry Shaw	2	1	2
9		Ryan Boggs	3	3	3
10	Judge-4	Jarek Kowalski	1	2	1
11		Michael C. Two	2	2	2
12		Ryan Boggs	3	3	3
13	Judge-5	Drieseng	1	1	1
14		Jarek Kowalski	2	2	2
15		Bernard Vander Beken	3	3	3
16	Judge-6	Ryan Boggs	1	3	1
17		Michael C. Two	2	2	2
18		Drieseng	3	1	3
19	Judge-7	Drieseng	1	1	1
20		Michael C. Two	2	2	2
21		Ryan Boggs	3	3	3
22	Judge-8	Drieseng	1	1	1
23		Michael C. Two	2	2	2
24		Ryan Boggs	3	3	3
25	Judge-9	Drieseng	1	1	1
26		Ryan Boggs	2	3	2
27		Scott Hernandez	3	3	3
28	Judge-10	Drieseng	1	1	1
29		Ryan Boggs	2	3	2
30		Jarek Kowalski	3	2	3

4.4. RESULTS

Our experiment had two goals: to determine both the heuristics and group weights as well as to evaluate the recommendation algorithm. Thus, in this section, we are concerned with showing the results of our experiment to achieve the two purposes. We first show the weights of the heuristics and groups that we applied to our algorithm according to their importance. Then, we discuss the accuracy of the algorithm in recommending experts by showing the results of the precision and the recall.

4.4.1. Heuristic and Group Weights

In this section, we show the weights that were generated from applying the Filtered Attribute Evaluations algorithm in Weka on the human judgments representing their importance and priorities. We explained in the Determining Heuristic and Group Weights Phase the four classifications (technical heuristics, social heuristics within the DSSD, social heuristics within the ERSF classification, and the three heuristic groups). Consequently, we discuss in this section the weights of each of these four classifications.

4.4.1.1. Technical Heuristics Weights

Figure 4.17 shows the weights of each technical heuristic. However, Type 1, Type 2, and Type 3 are considered one heuristic, but we separated them since we were also concerned with studying which of the developers who worked on those types might have better expertise and can better understand the code at hand. From our analysis of judge selections (section 4.3.2.1), we found that the developers who worked on Type-3 might have the best expertise since their changes to a code show that they might have good understanding of the logic of that code. However, we see in Figure 4.17 that Type-1 and Type-2 have higher weights, but these weights did not arise because of the importance of the types themselves but because some developers

worked on both types. In other words, if developers worked on two types and others worked on just Type-3, the former developers are considered to have higher expertise. Moreover, Type-2 received a higher weight than Type-1 because some judges think that the developers who have modified identifiers' names better understand what the code does.

Our analyses of other technical heuristics with type heuristic (ignoring the three subtypes), we found that judges think that the developers who have modified a large number of lines might be the ones who have good expertise. Moreover, among those developers if some of them have modified a code most recently or have worked on more than one fragment, then these developers are more likely to have better expertise than others. The type heuristic received less importance than other heuristics since from a human perspective finding developers who have worked on similar fragments might be enough to consider them as experts without considering which type they have cooperated on.

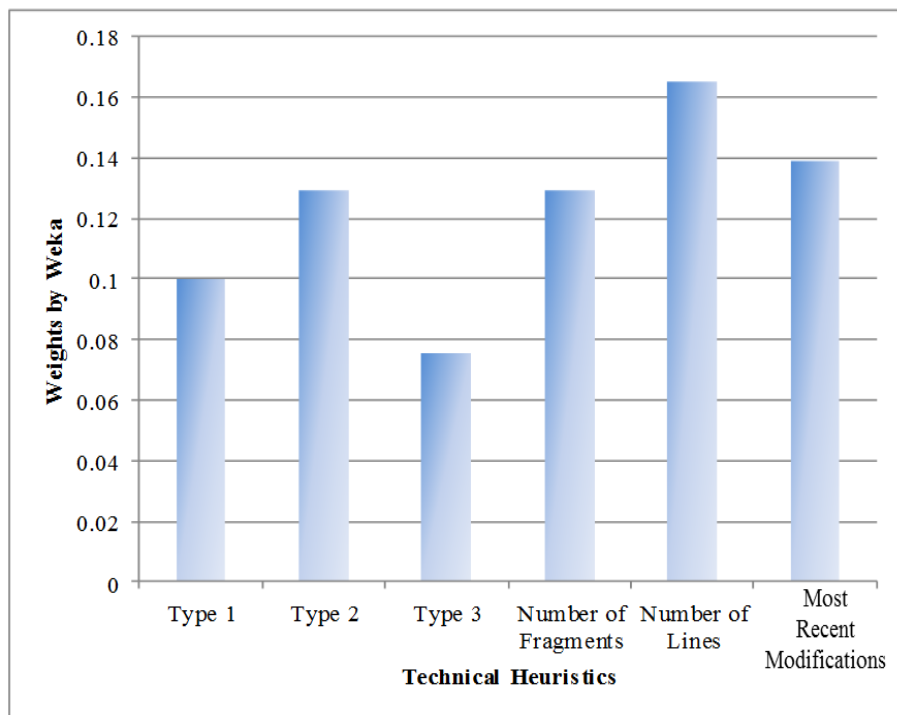


Figure 4.17: Technical Heuristics Weights Chart

4.4.1.2. Social Heuristics within the DSSD Weights

There are four heuristics under this group: two of them describe the developer relationships with the Active Developer, and the others describe the developer social relationships with others within the DSSD. Figure 4.18 shows the weights as analyzed by Weka.

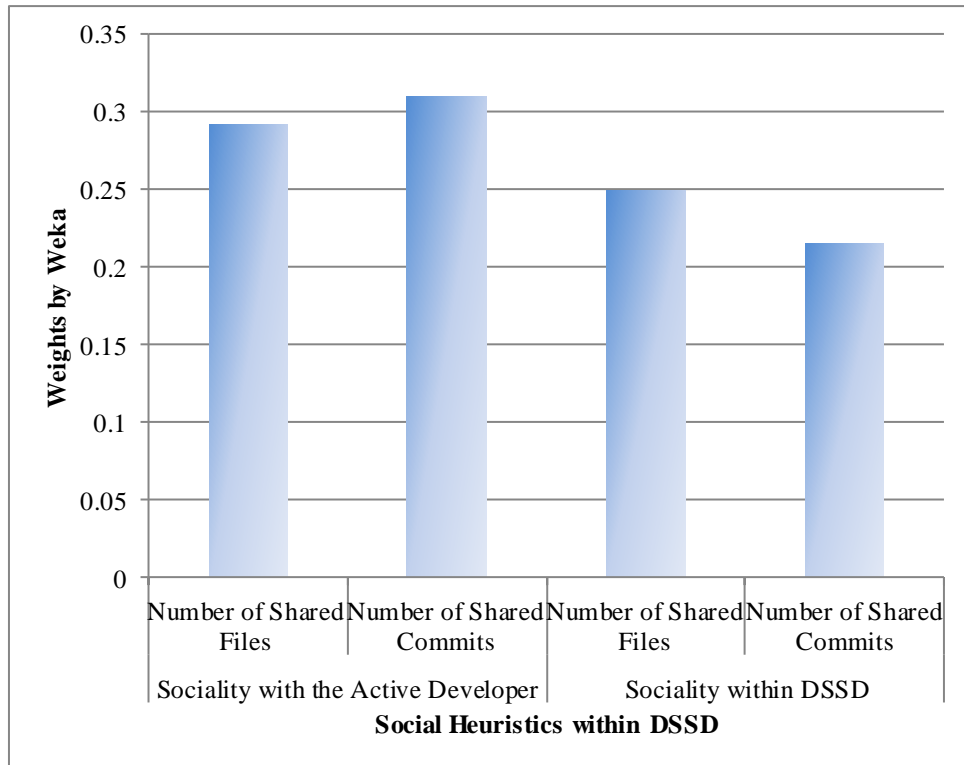


Figure 4.18: Social Heuristics within DSSD Weights Chart

Regarding this analysis, we found that some judges gave importance to the relationships in general without verifying what kinds of relationships they were. In other words, some judges considered the developers to be experts because the Active Developer has good relationships with them without looking at the basis of these relationships (counting number of shared files and commits) and/or because these developers have good sociality within the whole the DSSD (counting number of shared files and commits).

As a comparison of whether the developers who have social relationships with the Active Developer might be better social experts or the developers who are socially active within the DSSD, we see from Figure 4.18 that the first case, the social relationship with the Active Developer, is considered slightly more important to consider than the sociality within the DSSD.

4.4.1.3. Social Heuristics within ERSF Weights

Figure 4.19 shows the weights of the heuristics under this group, which were produced by Weka. As with the Social Heuristics within the DSSD, we found from our analyses that heuristics were mostly considered as relationships with the Active Developer or within the ERSF, ignoring the bases of these relationships (trust, response, help, and recommended). However, trust within the relationships with the Active Developer is given higher weights, followed by response, and then Help with the lowest weight. The reason behind this pattern is that since recommendation is dependent on help and help is dependent on response and so on, the amount of trust is always higher than the number of responses and so on, which caused the variation in weights between these four heuristics and not their importance. The same thing happened with the sociality within the ERSF; however, to have similar weights for the trust and responses within the ERSF shows that the response heuristic has higher importance than the others. Regarding the Recommendation with the Active Developer, its value is not dependent on any other heuristic; thus, to have the lowest weight mean that it has the lowest importance compared to other heuristics with the Active Developer; on the other hand, Recommendation within the DSSD has similar weight to the Helpfulness heuristic, which means it has similar importance as the Helpfulness heuristic.

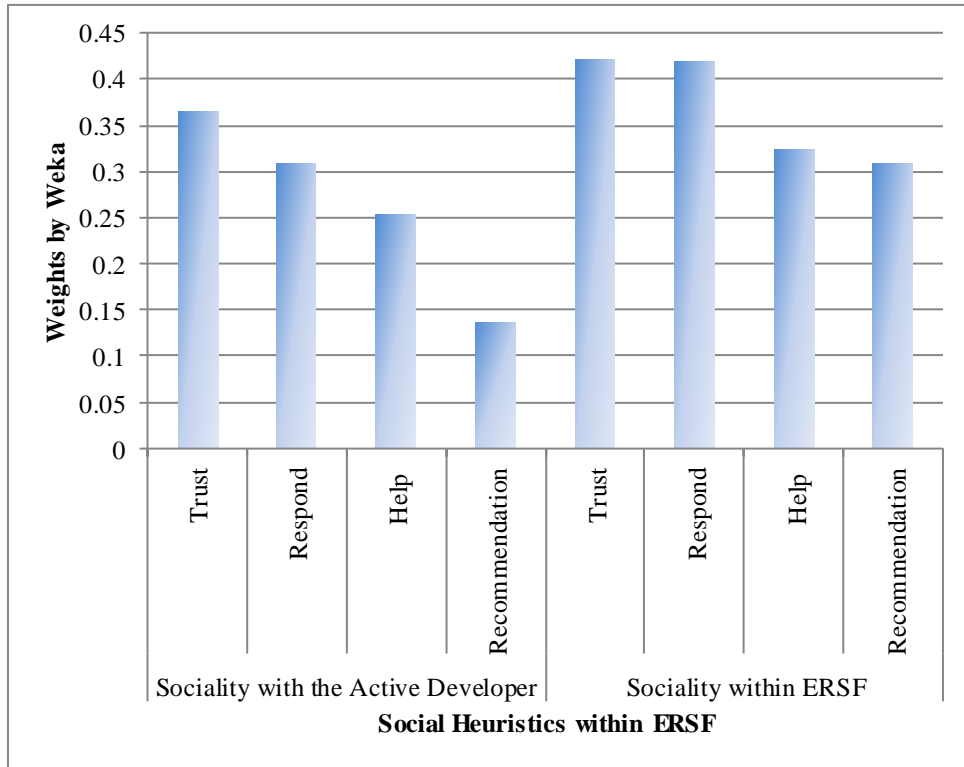


Figure 4.19: Social Heuristics within ERSF Weights Chart

4.4.1.4. Groups Weights

After we analyzed the importance of the heuristics among each other within each group, we also studied the importance of the groups themselves among each other. First, we analyzed the importance between the technical group and the social group within the DSSD. Then, we analyzed how using the recommender system and considering the social heuristics within the ERSF in identifying experts affects this importance and the priorities between the groups.

Figure 4.20 shows that human judges prefer to work and get help from social developers more than getting help from the developers who just have technical expertise. Thus, the social heuristics within the DSSD received more weight than the technical heuristics group.

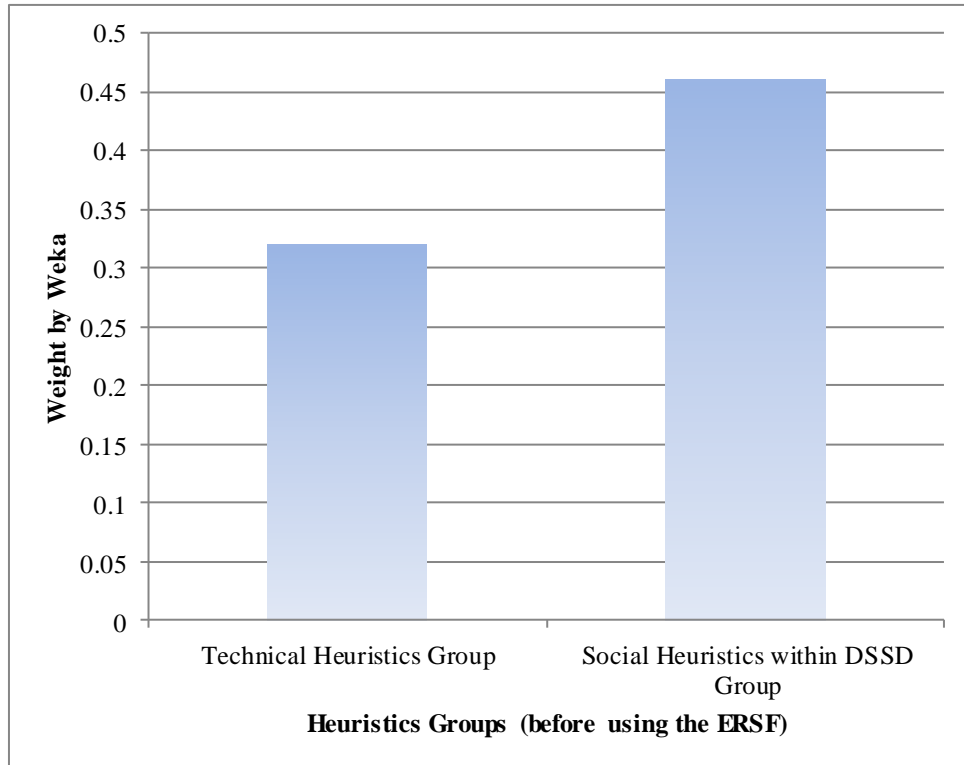


Figure 4.20: Groups Weights (Before Using the Recommender System) Chart

On the other hand, we see in Figure 4.21 that the social heuristic within the DSSD lost its importance when the recommender system was used. This shows that judges prefer to get assistance from the developers who are willing and helpful within the ERSF (social heuristics within the ERSF) more than getting help from the developers with whom they have worked with (social heuristics within the DSSD). Moreover, we see that the technical heuristics group has less importance than the social heuristics within the ERSF group, which suggests that judges still prefer to communicate and get help from the social developers more than the technical developers.

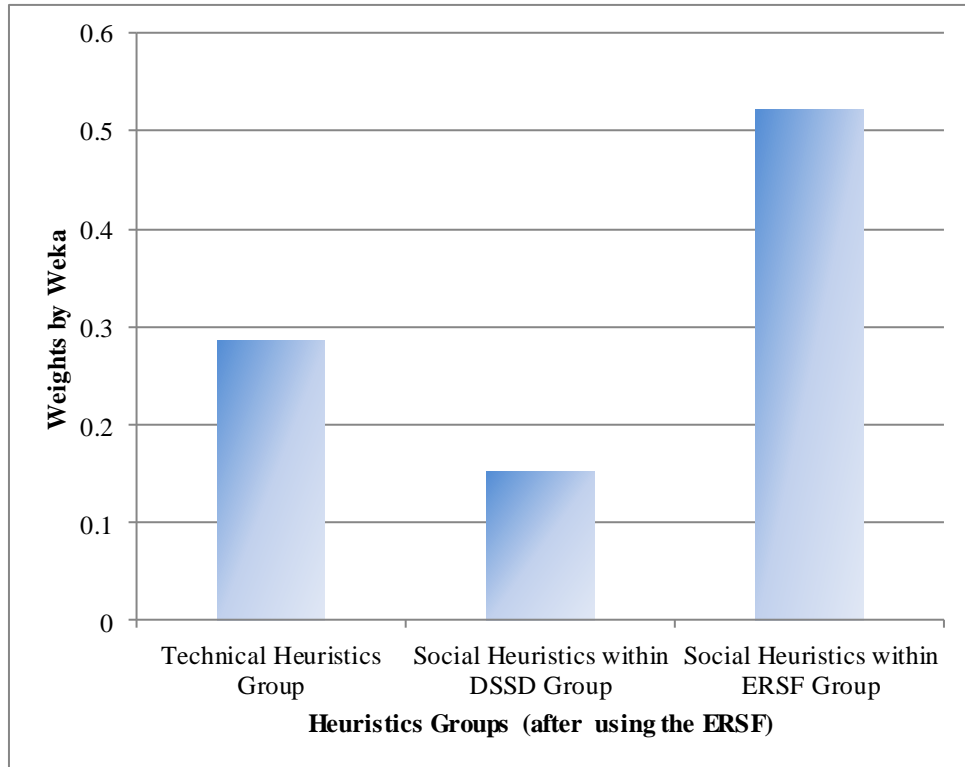


Figure 4.21: Groups Weights (After Using the Recommender System) Chart

4.4.2. Algorithm Accuracy

In recommendation system research, recommender system performance is evaluated by measuring the accuracy of its predictions and recommendations [13]. This is done by calculating the precision (representing the percentage of recommendations that are correct) and the recall (representing the percentage of correct experts recommended) [18].

In our experiment, we assume that the judge rankings and the predicted rankings by the NaiveBayes, NaiveNet, and J48 algorithms are the correct ones, on which we based the evaluation of our algorithm rankings. This is done by calculating the precision and recall to measure our algorithm accuracy. Figure 4.22 shows how we calculated the precision and the recall.

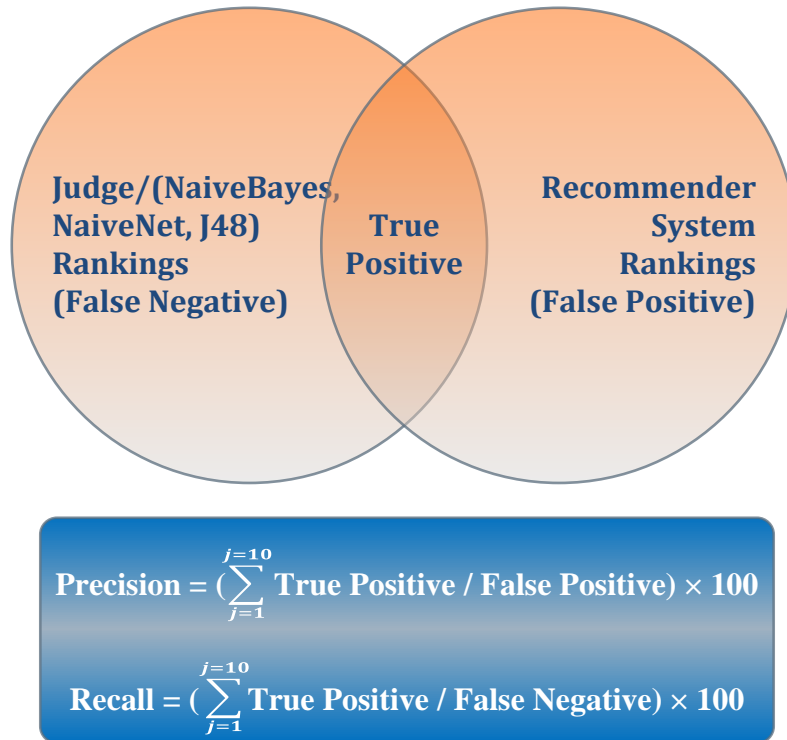


Figure 4.22: Calculating Precision and Recall

The precision and the recall equations are designed based on the following three measurements:

- i. False Positive:** contains the experts recommended by our algorithm but not by the judges.
- ii. False Negative:** contains the experts recommended by the judges but not by our algorithm.
- iii. True Positive:** contains the intersection of the algorithm recommendations and the rankings by the judges [13].

4.4.2.1. Comparisons of Scenario Rankings

In this section, we used every scenario comparison we discussed in the Evaluating the Algorithm Accuracy Phase in order to calculate the precision and the recall of our algorithm rankings. We first compared our algorithm rankings to the rankings by each judge/NaiveBayes in order to measure the True Positive of the comparison with this judge/NaiveBayes. Then, we

calculated the summation of the True Positive values of each comparison to find the Total True Positive that were then used to find the precision and the recall as shown below:

$$\text{Total True Positive} = \sum_{j=1}^{j=10} \text{True Positive of the Comparison with } j \quad \text{Formula-6}$$

In the following subsections, we explain how we applied this Total True Positive calculation in each scenario.

Regarding the False Negative, its values were always “30” since we had 10 judges and each judge ranked the top three experts, which in total are “30” rankings by the human judges; in addition, since the predictions of the NaiveBayes, NaiveNet, and J48 machine learning algorithms were based on the “30” judge rankings, which were restructured to “30” instances, we also had “30” rankings by the machine learning algorithms. Likely, the rankings by our algorithm were “30” rankings since they were identified based on the “30” rankings by the judges, which were also restructured to “30” instances to be our algorithm’s input, and this is what caused the False Positive to be “30” as well.

For these two reasons, we got the same precision and the recall in each scenario since the False Positive (The Precision denominator) and the False Negative (the Recall denominator) were both “30”. The measurement that presents our algorithm accuracy is the True Positive, which is concerned with the number of rankings our algorithm made that compatible with the judges/machine leaning algorithms.

A. Scenario-1 Ranking Comparison

Table 4.11 shows the comparison between our algorithm rankings and the judge rankings as well as the comparison between our algorithm rankings and the NaiveBayes rankings.

First, we measured the precision and the recall based on the comparison with the judge rankings. We compared our algorithm rankings to the rankings by each judge in order to measure

the True Positive. For example, when we compared our algorithm rankings (“Raya Boggs” with “2”, “Drieseng” with “3”, and “Scott Hernandez” with “1”) to the Judge-1 rankings (“Raya Boggs” with “1”, “Drieseng” with “2”, and “Scott Hernandez” with “3”), the resulted True Positive equals 1. Then, we calculated the summation of the Total True Positive values resulted from the comparison with each judge as follow:

$$\text{Total True Positive} = 0 + 0 + 1 + 3 + 0 + 3 + 0 + 1 + 1 + 3 = 12$$

The False Positive and the False Negative values are “30” as we mentioned earlier. Based on that the precision and the recall are calculated as follows:

$$\text{Precision} = (16 / 30) * 100 = 40\%$$

$$\text{Recall} = (16 / 30) * 100 = 40\%$$

Second, we measured the precision and the recall based on the comparison with the NaiveBayes rankings. We compared our algorithm rankings to the predicted rankings by NaiveBayes (which is equivalent to the rankings by each judge) in order to measure the True Positive. For example, when we compared our algorithm rankings (“Gerry Shaw” with “2”, “Ryan Boggs” with “3”, and “Scott Hernandez” with “1”) to the NaïveBayes rankings that are equivalent to the Judge-2 rankings (“Gerry Shaw” with “2”, “Ryan Boggs” with “1”, and “Scott Hernandez” with “3”), the resulted True Positive equals 1. Then, we calculated the summation of the Total True Positive values resulted from the comparison with each judge as follow:

$$\text{Total True Positive} = 1 + 1 + 2 + 2 + 0 + 0 + 0 + 1 + 0 + 0 = 7$$

The False Positive and the False Negative values are “30” as we mentioned earlier. Based on that the precision and the recall are calculated as follows:

$$\text{Precision} = (15 / 30) * 100 = 23\%$$

$$\text{Recall} = (15 / 30) * 100 = 23\%$$

B. Scenario-2 Ranking Comparison

Table 4.13 shows the comparison between our algorithm rankings and the judge rankings, as well as the comparison between our algorithm rankings and the NaiveBayes rankings.

First, we measured the precision and the recall based on the comparison with the judge rankings. We found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (19 / 30) * 100 = 63\%$$

$$\text{Recall} = (19 / 30) * 100 = 63\%$$

Second, we measured the precision and the recall based on the comparison with the NaiveBayes rankings. As in the comparison with the judge rankings, we found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (18 / 30) * 100 = 60\%$$

$$\text{Recall} = (18 / 30) * 100 = 60\%$$

C. Scenario-3 Ranking Comparison

Table 4.15 shows the comparison between our algorithm rankings and the judge rankings, as well as the comparison between our algorithm rankings and the NaiveBayes rankings.

First, we measured the precision and the recall based on the comparison with the judge rankings. We found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (30 / 30) * 100 = 100\%$$

$$\text{Recall} = (30 / 30) * 100 = 100\%$$

Second, we measured the precision and the recall based on the comparison with the NaiveBayes rankings. As in the comparison with the judge rankings, we found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (21 / 30) * 100 = 70\%$$

$$\text{Recall} = (21 / 30) * 100 = 70\%$$

D. Scenario-3 Group Ranking Comparison

Table 4.17 shows the comparison between our algorithm rankings and the judge rankings, as well as the comparison between our algorithm rankings and the NaiveBayes rankings.

First, we measured the precision and the recall based on the comparison with the judge rankings. We found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (21 / 30) * 100 = 70\%$$

$$\text{Recall} = (21 / 30) * 100 = 70\%$$

Second, we measured the precision and the recall based on the comparison with the NaiveBayes rankings. As in the comparison with the judge rankings, we found the Total True Positive of this comparison as in Scenario-1, and we then used it to measure the precision and the recall as follow:

$$\text{Precision} = (19 / 30) * 100 = 63\%$$

$$\text{Recall} = (19 / 30) * 100 = 63\%$$

4.4.2.2. Discussion

In this section we discuss the accuracy of our algorithm in recommending and ranking the experts as how much precisions and recalls we got from our algorithm comparisons to both the

human judges and to the NaiveBayes, NaiveNet, and J48 algorithms. Below we first discuss the results from the comparison to the judge rankings; then, we move to the results from the comparison with the NaiveBayes algorithm.

Table 4.18 summarizes the resulting precisions and recalls comparing the human judges and NaiveBayes algorithm in each scenario and the groups.

Table 4.18: Precision and Recall

Metric \ Comparison	Judges' Rankings		NaiveBayes's Rankings	
	Precision	Recall	Precision	Recall
Scenario 1	40%	40%	23%	23%
Scenario 2	63%	63%	60%	60%
Scenario 3	100%	100%	70%	70%
Groups	70%	70%	63%	63%

Human judges have inconsistent rankings among each other as shown in Table 4.11, 4.13, 4.15, and 4.17. However, our algorithm agreed on the three rankings (i.e. 1, 2, and 3) with two judges (Judge-4 and Judge-6) in Scenario-1; with five judges (Judge-2, Judge-4, Judge-7, Judge-8, and Judge-9) in Scenario-2; with all the judges (Judge-1 through Judge-10) in Scenario-3; and with seven judges (Judge-1, Judge-2, Judge-5, Judge-6, Judge-7, Judge-9, and Judge-10) in the group rankings. Moreover, the algorithm agreed with the judges in one ranking (i.e. 1, 2, or 3) with three judges (Judge-3, Judge-8, and Judge-9) in Scenario-1; with four judges (Judge-1, Judge-3, Judge-5, and Judge-10) in Scenario 2; and with three judges (Judge-3, Judge-4, and Judge-8) in the group rankings. This shows that our algorithm agreed, in total, with 83 of the human rankings out of 120, although there was inconsistency between the judge rankings.

Accordingly, our algorithm shows good to excellent precision and recall in its performance in Scenario-2, Scenario-3, and the group ranking comparison. However, we have low precision

and recall in Scenario-1 since not all the technical heuristics in this scenario are numeric as in the other scenarios; it also includes a date “Most recent modifications”, which might be the heuristic that affects our algorithm performance.

Likewise our algorithm agreed with the NaiveBayes algorithm in ranking four developers as “1st”, one developer as “2nd”, and two developers as “3rd” experts in Scenario-1; in ranking eight developers as “1st”, six developers as “2nd”, and four developers as “3rd” experts in Scenario-2; in ranking eight developers as “1st”, six developers as “2nd”, and four developers as “3rd” experts in Scenario-2; in ranking seven developers as “1st”, seven developers as “2nd”, and seven developers as “3rd” experts in Scenario-3; and in ranking seven developers as “1st”, seven developers as “2nd”, and seven developers as “3rd” experts in groups rankings. This means that our algorithm agreed, in total, with 67 of NaiveBayes’s rankings out of 120.

Accordingly, our algorithm shows good to excellent precision and recall in its performance in Scenario-2, Scenario-3, and the Group ranking comparison. However, we have low precision and recall in Scenario-1 since not all the technical heuristics in this scenario are numeric as in the other scenarios; it also includes a date “last date of modifications”, which might be the heuristic that affects our algorithm performance.

NaiveNet and J48 algorithms show the same precisions and recalls in Scenario-2, Scenario3, and the group comparisons as NaiveBayes. However, they were different than NaiveBayes in Scenario-1. First, for the NaiveNet, the precision and the recall were 33%. Second, for the J48, the precision and the recall were 30%. Our perspective on these differentiations is the effect of the “Most recent modifications” heuristic since this is the only scenario that uses the date not like other scenarios that were just concerned with numeric heuristics.

Overall, the good to excellent precision and recall compared to both the human judges and the NaïveBayes algorithm indicates that our algorithm, which considers both the technical expertise and sociality as well as being concerned with improving its performance during the use of the system, could be very useful to be used in the software system development organizations to help their developers finding the suitable experts who can help in the code at hand.

4.4.3. Limitations

The recommender system algorithm accuracy is affected by the heuristics and their group weights, which were determined based on the human judges. However, these human judges have faced difficulty making their decision in ranking the experts since they were given a great deal of data to consider. For example, in Scenario-3, the judges were given 19 developers with 12 characteristics representing the sociality of each developer, and they were given four characteristics representing the technical expertise of five of these developers; in total, they were given 248 characteristics to consider in their decision making. This problem affected the ranking accuracy of the judges. Thus, it affected the analyzing of the heuristics and group weights that represent their importance. Therefore, we plan to improve the method of collecting the human rankings by giving them more than three scenarios, but each scenario will include two developers with two heuristics representing their technical expertise and/or sociality. For instance, we will give the judges two developers with their characteristics that represent “Number of Lines” and “Trust” as shown in Table 4.19. This will make it easier for the judges to make their decision and to rank the developers; thus, we will get more accurate rankings by the judges and, consequently, more accurate weights to design the algorithm with better performance.

Table 4.19: Collecting Human Rankings

Developers Heuristics	Developer 1	Developer 2
Number of Lines	10	4
Trust	5	9

Another limitation is the number of judges we have run our experiment on. Applying machine learning algorithms requires a large amount of data to come up with better predictions. However, we only conducted our study using 10 judges with three rankings by each judge, so in total we had 30 rankings that produced 30 instances as input to these algorithms. This number of instances is considered low to be used as training data to learn from, analyze the patterns, and make the predictions. Thus, we plan to conduct our experiments using more judges.

Regarding the machine learning algorithms, we used the Filtered Attribute Evaluation attribute selection algorithm to analyze the heuristics and the groups in order to determine their weights that show their importance, but we did not try other algorithms like Gain Ratio Attributes, Info Gain Attributes, Relief Attribute Evaluation, and Symmetrical Attribute Evaluation algorithms that also assign weights to the heuristics as how much they are important, unlike other attribute selection methods that only rank the heuristics without assigning them any weights.

When we used the NaiveBayes, NaiveNet, and J48 classifiers, we tested the dataset (human rankings) with the “Use Training Set”. However, we found that the “Cross-validation” test is a better option to use since it divides the dataset into equally folded subsets and in every round it uses one subset as testing data and other subsets as training data ^{[4][5]}.

⁴ http://en.wikipedia.org/wiki/Cross-validation_%28statistics%29

⁵ <http://www.qsarworld.com/qsar-ml-cross-validation.php>

Finally, we will be providing our tool to the public so that developers can use it to find experts to help them. Therefore, we are planning to build an Eclipse plug-in for our recommender system.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. SUMMARY

Building successful software projects to work in a distributed software system development (DSSD) environment requires successful teams who can collaborate and cooperate to manage the complexity of these projects. The complexity of the projects comes from the dependency between their components; moreover, as the number of these dependent components becomes larger, the projects become more complex. Therefore, assigning developers to specific tasks should be done precisely; otherwise, it not only affects the performance of this task but also the performance of the dependent components. Since the team members need to collaborate in order to handle the complexity, any arguments between them might negatively affect the team's performance. This might lead to delay and/or failure of the projects.

To help ensure project success, in this thesis we developed an Expert Recommender System Framework (ERSF). The system recommends experts to the developers, Active Developers, when they need help completing code at hand based on technical and social measurements. It analyzes the developer expertise on similar code fragments to the code an Active Developer needs help with since someone who has worked on similar fragments might understand and be able to help with the current code; in addition, it analyzes the social relationships of the developers with the Active Developer and their social activities within the DSSD. Also, the system tracks the developers' communications through the ERSF and keeps the developer profiles up-to-date in order to improve its performance and recommendations each time the developers use the system.

Our system recommends experts based on their technical expertise and sociality, unlike other approaches, such as Codebook [4], CARES [7], Expertise Recommender [15], Emergent Expertise Locator [18], Expertise Browser [19], and Conscius [20] that limit their recommendations to be based on the technical expertise and Ensemble [30] that limit their recommendations to be based on the developer sociality within an organization. This is an advantage of our system since recommending experts who are not willing to help even if they have good knowledge and expertise in code or on the other hand recommending experts who are socially active developers but do not have adequate expertise in code will be useless.

Moreover, when considering social factors, besides analyzing the helpers' willingness to help, which is considered by STeP_IN [32,33], we also consider analyzing their ability to help since contacting developers who do not have ability to help even if they are willing to help is a waste of time. Another aspect we also considered in our work is assuring the comfort of both the Active Developer and the helpers to communicate since any disagreements between them might cause a project's failure, unlike STeP_IN that only considers the helpers' willingness.

In Chapter 3, we explained our methodology to recommend the experts. The developer expertise on the similar code fragments is measured using four technical heuristics: degree of code similarity, number of fragments, number of lines, and most recent modifications. We measure the sociality of the developers within the DSSD using four social heuristics: number of shared files and shared commits with the Active Developer as well as the number of shared files and shared commits within the entire DSSD. Finally, we measure the sociality of these developers within the ERSF using eight social heuristics: trust, response, help, and recommended to the Active Developers and within the ERSF.

In Chapter 4, we designed our experiment for two purposes. First, we are concerned with determining the heuristics weights both for the individual heuristics and the group weights (technical heuristics group, social heuristics within the DSSD group, and the social heuristics within the ERSF group) since not all the heuristics within a group, nor the groups themselves, have the same priorities and importance in recommending the experts. We applied the Filtered Attributes Evaluation method in the Weka Attribute Selection technique on the rankings by the human judges to the developers in order to determine the weights of the heuristics and groups as to how important they are in recommending the experts. Second, we designed this experiment to evaluate the accuracy of the recommending expert algorithm. We did this by conducting an analysis comparing the ERSF rankings to human judge rankings and to the rankings by NaiveBayes, NaiveNet, and J48 machine learning algorithms, as follows:

1. We compared our algorithm rankings to the rankings by the human judges.
2. We also compared the recommender system rankings to the predicted rankings by the machine learning algorithms.
3. Finally, we measured the algorithm accuracy by calculating the precision and the recall of the two comparisons. In the end, we obtained good to excellent precision and recall in this evaluation.

5.2. LIMITATIONS AND FUTURE WORK

Our work in conducting the experiment and designing the recommender system algorithm has some limitations we need to address in the future. Below we explain each of these limitations along with hints about how they could be fixed and improved.

5.2.1. Improving the Human Rankings Collection

The judges were provided with a long list of developers and a large amount of data representing their technical expertise and sociality to consider. This amount of data caused difficulty to the judges to make their decision in ranking the experts. As a result, this difficulty affected the accuracy of the rankings by the human judges. Consequently, the accuracy of assigning weights to the heuristics and the groups, which were then used in designing the recommender system algorithm, were also negatively affected.

Therefore, we would like to improve this experiment by providing more scenarios to the judges but with two heuristics to be considered and two developers to be ranked. This will help the judges make their decisions much more easily, and at the same time, we will have more accurate rankings to be analyzed to get more accurate weights.

5.2.2. Conducting the Experiment on More Judges

We ran our experiment using only 10 judges. However, part of our evaluation of the recommender system algorithm accuracy was based on using machine learning techniques. Such techniques require a large amount of data in order to provide more accurate predictions. Therefore, we need to run our experiment with data collected from more judges.

5.2.3. Try other Attributes Selection Methods in Weka

There are many attribute selection methods that could be used to determine the heuristics and group weights, such as Filtered Attribute Evaluation, Gain Ratio Attributes, Info Gain Attributes, Relief Attribute Evaluation, and Symmetrical Attribute Evaluation methods. These methods provide weights to the entered heuristics (attributes), unlike other methods that only rank these heuristics without providing any weights describing the importance of these attributes.

However, we just tried the Filtered Attribute Evaluation method since it seemed the best first method to apply that gives what we need.

5.2.4. Build an ERSF Plug-In

The proposed method is still in the prototype stage and a next step is to build a plug-in in Eclipse in order to make our system more widely available.

In conclusion, our recommender system has considered many important aspects that should be considered in recommending experts. It considers the technical expertise and sociality, as well as it considers improving its recommendation performance while the system is used. Moreover, our experiment shows that the recommender system algorithm has good to excellent accuracy by getting high precision and recall not only against the human judgements but also against the NiveBayes, NaiveNet, and J48 machine learning algorithms. These aspects indicate that our recommender system could be very useful in distributed software system development organizations to assist the developers finding the suitable experts to help with the code at hand.

REFERENCES

- [1] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 6, pp. 734–749, Jun. 2005.
- [2] O. Alonso, P. T. Devanbu, and M. Gertz, "Expertise identification and visualization from CVS," *Proc. 2008 Int. Workshop. Conf. on Mining Softw. Repos.*, Leipzig, pp. 125–128.
- [3] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty, "Supporting online problem-solving communities with the semantic web," *Proc. 15th Int. Conf. on WWW*, Edinburgh, 2006, pp. 575–584.
- [4] A. Begel, Y. P. Khoo, and T. Zimmerman, "Codebook: Discovering and exploiting relationships in software repositories," *Proc. 32nd ACM/IEEE Int. Conf. on Softw. Eng.*, Cape Town, 2010, pp. 125–134.
- [5] D. Billsus, C. A. Brunk, C. Evans, B. Gladish, and M. Pazzani, "Adaptive interfaces for ubiquitous web access," *Commun. ACM*, vol. 45, no. 5, pp. 34–38, May 2002.
- [6] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *Proc. 13th Int. Conf. Software Eng.*, Leipzig, 2008, pp. 481–490.
- [7] A. Guzzi, A. Begel, J. K. Miller, and K. Nareddy, "Facilitating enterprise software developer communication with CARES," *Proc. 28th IEEE Int. Conf. Softw. Maint.*, Riva del Garda, Italy, 2012, pp. 527–536.
- [8] W. Hill, L. Stead, M. Rosenstein, and G. Furnas, "Recommending and evaluating choices in a virtual community of use," *Proc. ACM Conf. Human Fact. Comp. Syst.*, Denver, CO, 1995, pp. 5–12.

- [9] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 952–970, Dec. 2006.
- [10] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Int. Comput.*, vol. 7, no. 1, pp. 76–80, Jan./Feb. 2003.
- [11] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert recommendation with usage expertise," *2009 IEEE Int. Conf. Softw. Maint.*, Edmonton, AB, 2009, pp. 535–538.
- [12] B. Macek, M. Atzmueller, and G. Stumme, "Profile mining in CVS-logs and face-to-face contacts for recommending software developers," *Proc. 2011 IEEE 3rd Int. Conf. Priv., Sec., Risk Trust / IEEE 3rd Int. Conf. Soc. Comput.*, Boston, MA, pp. 250–257.
- [13] D. W. McDonald, "Evaluating expertise recommendations," *Proc. ACM Int. Conf. Support. Group Work*, Boulder, CO, 2001, pp. 214–23.
- [14] D. W. McDonald and M. S. Ackerman, "Just talk to me: A field study of expertise location," *Proc. ACM Conf. Comput. Support. Coop. Work*, Seattle, WA, 1998, pp. 315–324.
- [15] D. W. McDonald and M. S. Ackerman, "Expertise recommender: A flexible recommendation system and architecture," *Proc. CSCW 2000 / ACM Conf. Comput. Support. Coop. Work*, Philadelphia, PA, 2000, pp. 231–40.
- [16] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? A study of coordination in a software project," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 307–324, May/Jun. 2011.

- [17] B. N. Miller, I. Albert, S. K. Lam, J. A. Konstan, and J. Riedl, "MovieLens unplugged: Experiences with an occasionally connected recommender system," *Int. Conf. Intell. User Interfaces Proc. IUI*, Miami, FL, 2000, pp. 263–266.
- [18] S. Minto and G. C. Murphy, "Recommending emergent teams," *4th Int. Workshop Mining Softw. Repos.*, Minneapolis, MN, 2007, pp. 12–13.
- [19] A. Mockus and J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise," *Proc. 24th Int. Conf. Softw. Eng.*, Orlando, FL, 2002, pp. 503–512.
- [20] A. Moraes, E. Silva, C. da Trindade, Y. Barbosa, and S. Meira, "Recommending experts using communication history," *Proc. 32nd ACM/IEEE Int. Conf. on Softw. Eng.*, Cape Town, 2010, pp. 41–45.
- [21] K. Nakakoji, Y. Ye, and Y. Yamamoto, "Comparison of coordination communication and expertise communication in software development: Motives, characteristics and needs," *New Front. Artificial. Int.*, Tokyo, 2009, pp. 147–155.
- [22] P. Resnick, P. Bergstrom, and J. Riedl, "GroupLens: An open architecture for collaborative filtering of netnews," *Proc. Conf. Comput. Support. Coop. Work*, Chapel Hill, NC, 1994, pp. 175–186.
- [23] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Method.*, vol. 17, no. 4, pp. 1–36, Aug. 2008.
- [24] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Softw.*, vol. 27, no. 4, pp. 80–86, Jul./Aug. 2010.
- [25] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," *Proc. 2008 Int. Workshop. Conf. on Mining Softw. Repos.*, Leipzig, pp. 121–124.

- [26] U. Shardanand and P. Maes, “Social information filtering: Algorithms for automating ‘word of mouth’,” *Proc. ACM Conf. Human Fact. Comp. Syst.*, Denver, CO, 1995, pp. 210–217.
- [27] S. Thummalapenta and T. Xie, “PARSEweb: A programmer assistant for reusing open source code on the web,” *Proc. 22nd IEEE/ACM Int. Conf. Automat. Softw. Eng.*, Atlanta, GA, 2007, pp. 204–213.
- [28] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, “On the effectiveness of simhash for detecting near-miss clones in large scale software systems,” *Proc. 18th Work. Conf. Reverse Eng.*, Limerick, Ireland, 2011, pp. 13–22.
- [29] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, Boston, MA: Elsevier, 2011.
- [30] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, S. X. Yang, “Ensemble: A recommendation tool for promoting communication in software teams. *Proc. 2008 Int. workshop Recommendation Syst. Softw. Eng.*, 2008, Article 2.
- [31] Y. Ye and G. Fischer, “Reuse-conducive development environments,” *Automat. Softw. Eng.*, vol. 12, no. 2, pp. 199–235, Apr. 2005.
- [32] Y. Ye, K. Nakakoji, and Y. Yamamoto, “Reducing the cost of communication and coordination,” *1st Int. Conf. Softw. Eng. Approaches Offshore Outsourced Develop.*, Zurich, Switzerland, pp. 152–169.
- [33] Y. Ye, Y. Yamamoto, and K. Nakakoji, “A socio-technical framework for supporting programmers,” *Proc. 6th Joint Meeting European Softw. Eng. Conf. and the ACM SIGSOFT Symp. Found. Softw. Eng.*, Dubrovnik, Croatia, 2007, pp. 351–360.

- [34] M. F. Zibran and C. K. Roy, “The road to software clone management: A survey,” Dept. Comput. Sci., Univ. of Saskatchewan, Saskatoon, SK, Tech. Rep. 2012-03, Feb. 2012.
- [35] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

APPENDIX A

HUMAN RANKING COLLECTION

In this section we provide the list of developers with their technical expertise and sociality that we provided to the judges in our experiment as explained in section 4.3.1. The first table lists the developers provided in Scenario-1 (section 4.3.1.1), the second table lists the developers provided in Scenario-2 (section 4.3.1.2), and the last table lists provided in Scenario-3 (section 4.3.1.3).

1. Developer Technical Expertise List

Developer Names	Clone Types	Number Of Fragments	Number Of Lines	Most Recent Modification
Ryan Boggs	Type-2	2	19	2012-05-04
Drieseng	Type-1 Type-3	3	17	2004-08-11
Scott Hernandez	Type-1 Type-2	2	13	2003-12-08
Gerry Shaw	Type-3	1	12	2002-08-14
Michael C. Two	Type-1	1	9	2002-10-25

2. Developer Sociality within the DSSD List

Developer Names	Sociality with You		Sociality within the DSSD	
	Number of Shared Files	Number of Shared Commits	Number of Shared Files	Number of Shared Commits
Ryan Boggs	0	0	0	0
Dguder	0	0	2	6
Rmboggs	1	2	2	4
Charles Chan	10	23	10	23
Drieseng	10	20	30	60
James Geurts	4	9	10	54
Ian MacLean	2	6	2	6
Giuseppe Greco	0	0	20	64
Scott Hernandez	8	14	10	17
Gerry Shaw	0	0	20	78
Michael C. Two	0	0	0	0
Matthew Mastracci	4	16	10	34

Developer Names	Sociality with You		Sociality within the DSSD	
	Number of Shared Files	Number of Shared Commits	Number of Shared Files	Number of Shared Commits
Jarek Kowalski	4	23	14	54
Claytonharbour	0	0	21	43
Bernard Vander Beken	0	0	0	0
Dmitry Jemerov	0	0	0	0
Owen Rogers	0	0	0	0
Dominik Guder	0	0	0	0

3. Developer Sociality within the ERSF List

Developer Names	Sociality with You				Sociality within the ERSF			
	Trust	Response	Helpfulness	Recommended	Trust	Response	Helpfulness	Recommended
Ryan Boggs	0	0	0	0	26	23	18	20
Dguder	0	0	0	0	0	0	0	0
Rmboggs	0	0	0	0	0	0	0	0
Charles Chan	0	0	0	0	0	0	0	0
Drieseng	15	11	9	2	36	25	15	19
James Geurts	0	0	0	0	0	0	0	0
Ian MacLean	0	0	0	0	0	0	0	0
Giuseppe Greco	0	0	0	0	0	0	0	0
Scott Hernandez	0	0	0	0	0	0	0	0
Gerry Shaw	0	0	0	0	0	0	0	0
Michael C. Two	18	15	9	5	19	17	10	7
Matthew	0	0	0	0	0	0	0	0

Developer Names	Sociality with You				Sociality within the ERSF			
	Trust	Response	Helpfulness	Recommended	Trust	Response	Helpfulness	Recommended
Mastracci								
Jarek Kowalski	17	13	8	21	26	20	15	32
Claytonharbour	5	4	1	5	9	6	1	7
Bernard Vander Beken	9	7	5	8	12	11	9	12
Dmitry Jemerov	0	0	0	0	21	18	14	21
Owen Rogers	2	2	1	0	9	5	2	4
Dominik Guder	1	0	0	0	1	0	0	0