Recommending Software Experts Using Code Similarity and Social Heuristics

Ghadeer A. Kintab[+] Chanchal K. Roy Gordon I. McCalla
[+]Ministry of Higher Education, Saudi Arabia
Department of Computer Science, University of Saskatchewan, Canada
gh_kintab@yahoo.com, {croy, mccalla}@cs.usask.ca

Abstract

Successful collaboration among developers is crucial to the completion of software projects in a Distributed Software System Development (DSSD) environment. We have developed an Expert Recommender System Framework (ERSF) that assists a developer (called the "Active Developer") to find other developers who can help them to fix code with which they are having difficulty. The ERSF first looks for other developers with similar technical expertise, as measured by their prior work on code fragments that are similar to (clones of) the code that the Active Developer is working on (the "code at hand"). As well, it analyzes the other developers' social relationships with the Active Developer (available from the DSSD environment) and their social activities within the ERSF (information which helps to maintain developer profiles used in this analysis). This information is then combined to provide a ranked list of potential helpers based on both technical and social measures. A proof of concept experiment shows that the ERSF can recommend experts with good to excellent accuracy, when compared with human rankings of appropriate experts in the same scenarios

1 Introduction

A software system is a composite of dependent components that make the software complicated especially if the system has a large number of such components; a single developer may have limited knowledge, and s/he is unable to work on all the components [5]. Therefore, developers need to cooperate and coordinate to manage system dependencies and build a successful software system. 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. [1] found that most Microsoft software engineers need on occasion to find relevant engineers to help them accomplish tasks at hand.

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 [15].

There have been a great many studies in finding expert developers as well [7,12,21] (for details see Section 2). Among different metrics for recommending experts, the most important are the technical expertise of the potential helpers on the code fragment the Active Developer is working on and the social relationships among the Active Developer and the other developers. However, most previous techniques to recommend experts are based only on their technical expertise. Such techniques have a potential danger: if developers are not very skilled at collaboration and/or 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 is done in *Ensem-*

ble [19], will be useless if the recommended developers do not have adequate knowledge of the code even if they have strong social relationships. Although there is at least one expert recommender system that exploits both the technical expertise and social relationships of the developers (STeP_IN [21]), our approach breaks new ground both in how we measure technical expertise and how we track social relationships.

We developed an Expert Recommender System Framework (ERSF) to identify and allocate a ranked list of expert helpers who can help an Active Developer who is looking for assistance in completing or fixing a buggy code fragment. The system considers both technical expertise and social relationships of developers in recommending those experts. The technical expertise is concerned with finding people who have worked on code fragments in the past that are similar to the code being developed by the Active Developer. These are called clones in the Software Engineering area [14]. In detecting clones we use a state of the art clone detector, SimCad [17] that detects both exact and near-miss code-fragments (similar fragments where there could be differences among them in terms of number of statements, identifiers, comments and formatting). This provides a much larger range of potential helpers with appropriate expertise than in previous systems that mostly determine the technical expertise of the other developers based on selecting developers who have worked on exactly the same code for which the Active Developer is seeking help.

The social relationships are concerned with the social activities between the Active Developer and other developers during Distributed Software System Development (DSSD). We developed further measures of ability, trust, and reputation by analyzing developer communications within the Expert Recommender System Framework (ERSF). This allows the ERSF to create profiles of each developer that, over time, can be maintained and updated. This is an improvement over other methods (e.g., [15]) that base social measurements on data in a repository or API library.

Our method has other advantages, too. It can still provide recommendations, even if the code at hand has no authors other than the Active Developer himself/herself and even if there are no clones. It does this by using the social part of the system. On the other hand, developers with no or little history in the DSSD or ERSF can also get recommendations since our approach can use information about clones and/or general sociality within the DSSD and the ERSF. Further, developers can also get help from expert developers, even when completing code in new files using information about clones from other parts.

We have evaluated our approach with a proof of concept experiment with three different scenarios, 10 human judges and three machine learning algorithms [18]. Our experiment shows that the approach can recommend experts with comparable accuracy to the human judges.

The paper is organized as follows: Section 2 shows related work in recommender systems. Section 3 explains in detail the method we developed to recommend experts. Section 4 describes our experiment and the evaluation details. Section 5 points out some potential threats and then discusses how we mitigate them, and finally Section 6 concludes with our future plan.

2 Related Work

Recommending experts in software development is not new and there have been a great many studies. Expertise in recommender systems is usually judged using five different techniques. *Time-based techniques* [9,10] rank as highest those developers who worked most recently on a code fragment since they have the freshest code in mind. *Modification-based techniques* analyze added, removed, or edited lines of code to identify expertise [12,6,21]. Moreover, some systems benefit from commits on code besides the lines of code themselves to identify expertise [1], or sometimes they limit the identification based on the checked-in commits on code [3]. Developer expertise is measured based on the size of these modifications or commits. *Code usage-based techniques*, on the other hand, do not analyze the implementation of code to identify expertise; instead, they identify this expertise based on their knowledge about calling or using methods [15]. *Dependency-based techniques* [12,19] consider developers who have worked on the same or dependent artifacts to the artifact at hand who have knowledge about these artifacts and can help on the current artifact. Also, these techniques suggest that developers who have cooperated on the same or dependent artifacts should communicate with each other. Finally, *similarity-based techniques* find context similarity between artifacts and identify experts as those who have cooperated on those similar artifacts [13].

Most of the previous techniques recommend experts based only on their technical expertise as exhibited by the code they have developed. This could be problematical, as some of these experts might not be willing to help. Similarly, just considering the social relationships (e.g., as in *Ensemble* [19]) might not be useful if they do not have technical expertise on the target code fragment.

In order to address these problems, both technical expertise and social relationships should be considered when designing recommender systems. STeP_IN [21] has considered both the technical and social aspects of a recommendation. However, in term of sociality, STeP_IN looks at the willingness of the helpers to help but not their ability to help. 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 Developer's side, i.e., if the Active Developer is also willing to contact the helpers for assistance, if they trust the helpers, 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 ensure that they both are comfortable contacting each other and working with the other party and that their communications will not cause any failure during project development.

Our work is different from the above research in the sense that we use a hybrid method where we use both the technical and social heuristics in innovative ways. In particular, for gathering a large group of technical experts, we not only gather the developers who worked on the code fragment being developed by the Active Developer, but also all the developers who contributed cloned fragments that are similar to the one being worked on. We determine similarity using a state of the art clone detection tool that takes into consideration all three types of clones. Similarly, for the social heuristics, we also use new measures gathered from the version control systems of both the DSSD and ERSF. Furthermore, our approach can recommend experts even for new code under development by using the cloning technology and social heuristics, thus helping with cold start problems.

3 Developed Architecture and Methodology

In this section, we give a detailed explanation of how each technical and social heuristic is designed to measure the expertise and sociality of each developer within the DSSD and the ERSF. After that, we explain how we combine them to measure the likelihood of a developer to be the right expert to help the Active Developer. We conclude by describing the overall design and architecture of the ERSF.

3.1 Identifying Experts

An expert is defined relative to the purpose of a study or a developed system. In our work, we define an expert as a developer, other than the Active Developer, who has knowledge of the code at hand and/or of clones of that code, has good social collaboration with the Active Developer and/or within the organization, or has both the knowledge and the sociality.

3.1.1 Technical Heuristics

We assume in the technical part of the ERSF that developers who have worked on cloned fragments might understand the current code better and can help complete it. Also, we assume that if the current code is written or modified by developers other than the Active Developer himself/herself, these developers might be good helpers as well. However, these experts might have different degrees of expertise, which is determined and measured using the following heuristics:

- **Degree of Code Similarity (Clone Type):** detecting only exact clones does not always help, as the Active Developer may have an incomplete code fragment or the target code fragment may have significant dissimilarity with the existing code in the system. This is why we look for clones of type 1, 2, and 3 that allow for increasingly diverse expressions of the same functionality (for details, see Section 3.3.). Therefore, this measure is based on which type of clone a developer has worked on.
- **Number of Fragments:** this heuristic is measured based on the number of clone fragments a developer has worked on.

• **Number of Lines:** this heuristic assumes that as the number of lines a developer has modified increases, the developer gains more expertise.

• **Most Recent Modifications:** McDonald and Ackerman [9] identify expertise as belonging to the person who has modified a piece of code most recently since s/he is the one with the freshest code in mind. We use this heuristic as one of the measurements in our work.

3.1.2 Social Heuristics

The social heuristics analyze the relationships of the developers to the Active Developer and their social activities within the DSSD and the ERSF as follows:

a) Social Heuristics within the DSSD: A repository has a great deal of valuable information about the developers from which we can benefit. In our approach, we analyze the developer activities in the Git repository and construct the social relationships between developers within the DSSD. These relationships 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 (have relationships with other developers other than the Active Developer) within the DSSD. Therefore, we design the following heuristics to measure various aspects of the relationships between the Active Developer and other developers within the DSSD, as well as the degree of their social activity:

• **Number of Shared Files with the Active Developer:** a developer is considered closer to the Active Developer as the number of files they have shared increases.

• **Number of Shared Commits with the Active Developer:** a developer is closer to the Active Developer as the number of commits they have shared increases.

• **Number of Shared Files within the DSSD:** a developer gains more sociality within the whole DSSD as the number of files s/he shared with others increases.

• **Number of Shared Commits within the DSSD:** a developer gains more sociality within the whole DSSD as the number of commits s/he shared with others increases.

b) Social Heuristics within the ERSF: We are interested in further improving our recommendations to the developers by using information gathered as they use our ERSF. This is done by tracking the developer communications when they use the ERSF and keeping their profiles up-to-date. Moreover, we also design other social heuristics to improve the system performance and apply them to the developer profiles to measure their sociality. Below we provide the details:

• **Trust of the Active Developer in Others:** the number of times the Active Developer has trusted a developer determines how close this developer is to the Active Developer. The trust is determined by capturing who the Active Developer chose to get help from in the past; we assume that the developer that the Active Developer has chosen is trusted by the Active Developer.

• **Response to the Active Developer:** a developer is closer to the Active Developer as the number of his/her responses to the Active Developer increases.

• **Developers who Have Helped the Active Developer:** the more times a developer helps the Active Developer the closer this developer is to the Active Developer.

• **Recommended Developer to the Active Developer:** Begel et al. [1] 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 to the recommended developers the ability to recommend others if they are not able to help the Active Developer on his/her request. The system uses this heuristic as one measurement of the developer closeness to the Active Developer.

• **Developer Trust within the ERSF:** a developer gains in their sociality to the extent that s/he was trusted by others.

• **Developer Response within the ERSF:** a developer's response to others gives her/him more sociality.

• **Developer Helpfulness within the ERSF:** the number of times a developer has successfully helped other developers (not just the Active Developer) in the past determines how active this developer is.

• **How Often Developers are Recommended within the ERSF:** the recommendations of developers as experts by others identifies the overall reputations of those developers within the ERSF.

A developer gains more reputation insomuch as s/he was recommended by others.

3.2 Measuring a Developer's Likelihood to be an Expert

Each developer's likelihood to be a suitable expert is measured using a combination of all the heuristics explained in the "Identifying Experts" subsection using the following formula (1):

$$D_e = \sum_{h=1}^{n} w_{g(h)} \left(\frac{D_{(e/s)}}{T_{(e/s)}} \times w_h \right) \quad 0 \le D_e \le 1 \qquad (1)$$

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, $w_{g(h)}$ 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 w_h is the heuristic weight. We will explain how we assign each heuristic and each group weight in Section 4.

The algorithm will be explained first by looking at one technical/social heuristic. 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):

$$D_e = \frac{D_{(e/s)}}{T_{(e/s)}} \qquad (2)$$

For example, consider the Number of Lines heuristic, and assume that we have a piece of code with 15 lines (T_e). Three developers D_1, D_2, and D_3 have collaborated on the modification of this code as follows: D_1 has written 4 lines, D_2 has written 8 lines, and D_3 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 (2) on each developer as shown in Table 1.

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

Table 1: Developer Likelihood to be an Expert Example (Formula 2)

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

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

Further, assume that we have a Trust heuristic between the above developers (D_1, D_2, and D_3), besides the Number of Lines heuristic. Table 2 shows the expertise and sociality of the three developers 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.

Heuristics \ Developers	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	-

Table 2: Developer Likelihood to be an Expert Example (Formula 3)

Another important aspect of having more than one heuristics 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 the decisions of human judges. We have conducted an experiment to extract these 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 considered while they were ranking the developers. After that, we used the Weka tool [20], which is a collection of machine learning algorithms for data mining task. These algorithms are applied to a dataset to analyze its structural patterns, and make some predictions [4]. Weka is used in our work to analyze the judge rankings and come up with the heuristic weights within a particular group (this is explained in detail in Section 4). Based on this, we developed our algorithm to consider the heuristic weights (w_h) as in formula (4):

$$D_e = \sum\nolimits_{h=1}^{n} \left(\frac{D_{(e/s)}}{T_{(e/s)}} \times w_h \right) \qquad (4)$$

Moreover, since we have three groups, it is also desired to analyze the priorities and importance of each group overall, when compared to one another. Thus, we used both the judges' rankings and the Weka tool as well to determine the group weights ($w_{g(h)}$) based on their importance. As a result, we improved our algorithm in formula (1). 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 Section 4.

3.3 Experts Recommender System Architecture

Our ERSF approach is designed on top of the SimCad Clone Detection tool to extract the clones. Software clones or duplicated fragments of code in a software system are one of the important aspects of software systems as software developers often reuse code fragments by copying and pasting with or without minor adaptations [14]. Research shows that a significant fraction of code in software systems is cloned code [14,16]. Considering the fact that there are lots of clones in software systems and that multiple authors could be involved in those cloned fragments, in this paper, we used cloning in an innovative way, finding the authors who worked on similar fragments and then infer the experts for such similar fragments. However, in our case, detecting only exact clones does not address the problem at hand, as the active developer may have an incomplete code fragment or the target code fragment may have significant dissimilarity with the existing ones in the systems. We thus adapted a state of the art clone detection tool, SimCad that finds three different types of clones as follows:

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

Another tool we used in designing the recommender system is the Eclipse Communication Framework/DocShare plug-in (ECF/DocShare) [2] to provide a channel to the developers to communicate with the recommended experts. The plug-in allows two developers in a distributed location to share their editors in order to collaborate to write or modify the shared code. At the same time, both of these developers can have a conversation through the provided chat. Figure 1 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 plug-in. 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 by ranking 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.

3.3.1 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.

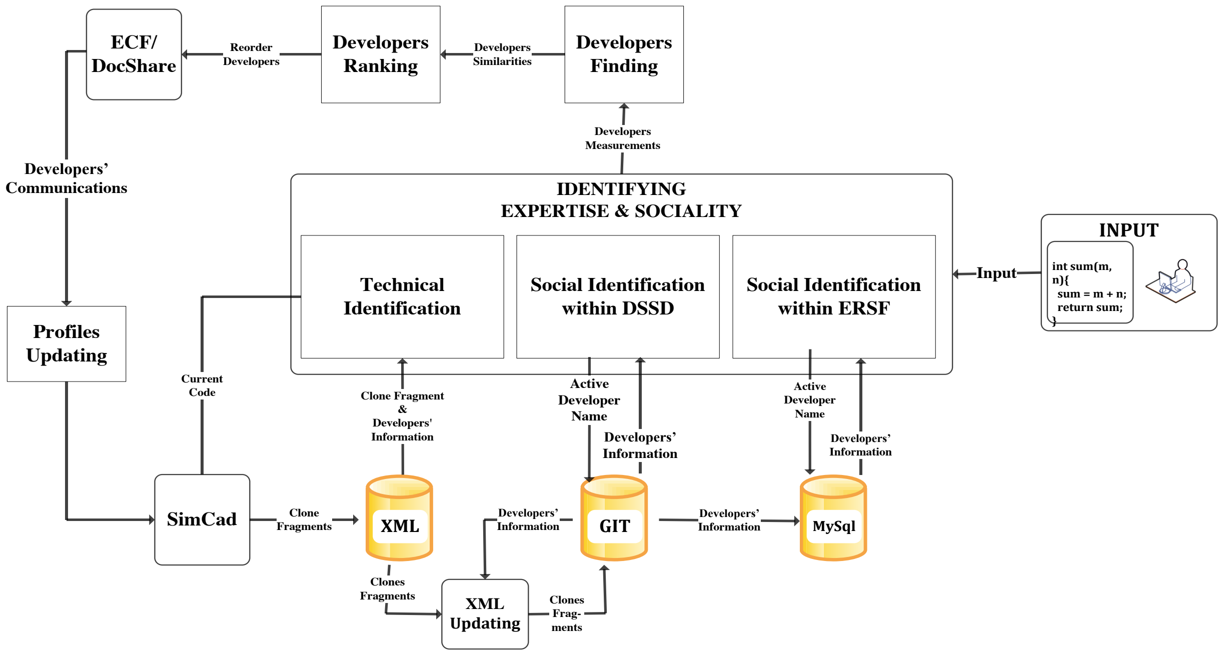

Figure 1: ERSF Architecture

a) Technical Expertise Identification: The technical expertise identification component sends the source code input (the code at hand on which the Active Developer is working) to the SimCad tool, which finds the clone fragments and saves them in an XML file. This file is then updated in the "XML Updating" component to include information about developers who have worked on those cloned 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) and measures their expertise using the technical heuristics.

b) Social Ability Identification within the DSSD: Identifying social ability within the DSSD component takes the Active Developer’s name and extracts his/her communications with other developers. It also extracts the developers' communications with each other. These data are extracted from the Git Repository used to design the social heuristics in order to measure the developers’ relationships to the Active Developer and their overall sociality within the DSSD.

c) Social Ability Identification within the ERSF: As with the previous component, this component takes the Active Developer’s name and extracts his/her communications with other developers, as well as the developers' communications with each other. This component extracts the data from the RS MySql database, which has the tracked communications within the ERSF. The purpose for this component is to measure the developers’ relationships with the Active Developer and their overall sociality as in the previous component but using the social heuristics that are measured when they were helping (or being helped) through our ERSF.

3.3.2 Finding Experts

After the system measures the developer expertise in the cloned 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. Then, using the equations in Section 3.2, we compute their likelihood to be suitable experts to help the Active Developer.

3.3.3 Ranking Developers

The main goal of this system is to recommend a ranked list of developers. Therefore, this component reorders the given list of developers in the ECF/DocShare to display them in a ranked list according to their likelihoods, from the previous component, to be suitable experts to help the Active Developer.

3.3.4 Updating Profiles

Through the ECF/DocShare plug-in 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 in the "Social Identification within the ERSF" component when any developer needs experts to contact.

4 Experiment and Evaluation

In our work we came up with 16 heuristics to measure the developer technical expertise and/or sociality, as we explained in Section 3. However, not all of these heuristics and the groups have the same degrees of importance in measuring expertise and/or sociality. To determine what the weights should be on the 16 factors we carried out an experiment in which we created three scenarios in which software engineers needed help. We asked 10 human judges who they thought would be the best people to recommend in each scenario. We then used machine learning algorithms (from the Weka toolkit) in two ways: to find appropriate weights for each factor and to determine how accurate we were in making recommendations that were comparable to the human judgments. The experimental process is summarized below. More details can be found elsewhere [4].

4.1 Experimental Methodology

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.

4.1.1 Collecting Human Rankings Phase

Our experiment was built based on human judgments. We ran the experiment using 10 judges who were 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 primarily with one group of heuristics (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF). We wanted the judges to go through the scenarios in order, considering the heuristic group of the previous scenario while they were making their decision in the next scenario. Thus, we included the data from the previous scenario to be considered in the current scenario. For instance, if the judges were working on Scenario-2, which is mainly concerned with judging sociality within the DSSD, we still wanted them to consider factors about developer expertise from the technical judgments, which are the main 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 each of our judges to assume that they were the Active Developer who is 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. After that, we asked the judges to rank the developers they thought were the best experts to contact and get help from (choosing a first, second, and third candidate). Also, we asked them to indicate the reasons for their selections; the reasons they could select from were the same heuristics we suggested for our algorithm.

At the end, the human rankings consisted of three elements: the developers chosen by the judges, the heuristics that the judges felt they had used in making these choices, and the rankings (i.e. 1, 2, or 3) of these developers. These elements were then used as follows: 1) Both the heuristics and the given rankings were analyzed to find out the heuristics and group weights to tune the recommender system algorithm, and 2) the three elements together (the developers chosen by the judges, the heuristics they used, and the rankings of these developers) were used to evaluate the accuracy of the recommender system algorithm once tuned.

4.1.2 Determining Heuristic and Group Weights Phase

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 degrees of importance. We applied the Filtered Attribute Evaluation method from the Weka Toolkit, which is a specific attribute selection technique, to the selected technical heuristics of Scenario-1, to the selected DSSD social heuristics of Scenario-2, and to the selected ERSF social heuristics of Scenario-3 to find out their weights. After this, we applied the same method to all 16 heuristics combined (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 relative weight and importance of each group.

Our analysis in this phase was done on the rankings of each scenario as follows: First, for each judge and for each of his/her rankings, we extracted the heuristics s/he considered while ranking a particular developer. For example, Table 3 shows the rankings by Judge-5. Second, we replaced those heuristics with the corresponding values that represent the ranked developers' expertise/sociality, depending on the scenario being worked on. For instance, "Charles Chan" was ranked second by Judge-5 because of his sharing in the files and the commits with the Active Developer as shown in Table 3, so we replaced these two heuristics with the number of files (i.e. 10) and the number of commits (i.e., 23), that "Charles Chan" has shared with the Active Developer. We also extracted the ranking the judge gave to the developer in order to analyze the importance of these heuristics from the judge's perspective.

The above data were then used to create an instance representing the judge's decisions to be used as input to the Weka tool. Table 4 represents a Weka input instance, which includes the values

Judges	Rankings	Developers	Considered Technical Heuristics
...			
Judge-5	1	Scott Hernandez	**Sociality with the Active Developer:** ...
	2	Charles Chan	**Sociality with the Active Developer:** • Number of shared files • Number of shared commits
	3	Drieseng	**Sociality with the Active Developer:** ... **Sociality within the DSSD:** ...
...			

Table 3: Scenario-2 (Judge-5 Ranking`s)

Values	Social Heuristics within the DSSD
<instance>	
<value>10</value>	Number of shared files with the Active Developer
<value>23</value>	Number of shared commits with the Active Developer
<value>0</value>	Number of shared files within the DSSD
<value>0</value>	Number of shared commits within the DSSD
<value>2</value>	Ranking
</instance>	

Table 4: Social Heuristics within the DSSD (the Filtered Attribute Evaluation Method Input Example)

of the heuristics that were selected by Judge-5 in making "Charles Chan" the second ranked developer.

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 relative to the values returned by the other heuristics.

4.1.3 Designing the Recommender System Algorithm

In this section, we provide an example to show how we used the weights that resulted from the Weka analysis (to prioritize the heuristics and groups) to actually compute the developer likelihoods to be experts who could also be socially able to help the Active Developer. This is the basis of the ERSF.

Lets say 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 trusts, 18 out of 128 responses, 14 out of 84 helpfulness, and 21 out of 126 recommended within ERSF. From the Weka analysis, we came up with the weights in Table 5.

Groups	**Heuristics**	**Heuristic Weights (w_h)**	**Group Weights ($w_{g(h)}$)**
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	

Table 5: Heuristic and Group Weights by Weka Example

We then applied our algorithm using formula (1) to each of the two developer characteristics to compute their likelihood to be an expert as follows:

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

Dmitry Jemerov = [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.

4.1.4 Evaluating the Accuracy of the Algorithm Phase

Another major goal in our experiment was to evaluate the accuracy of our algorithm in recommending suitable experts to assist the Active Developer in completing the code at hand. We did this for each scenario by comparing our algorithm's rankings to the judges' rankings using the NaiveBayes, NaiveNet, and J48 classifiers in Weka. We used these three classifiers in our experiment since we needed algorithms that predict rankings based on independent numeric attributes. In this phase we were not concerned with tuning the weights of our algorithm, as in our first phase, but were concerned only with the accuracy of our tuned algorithm and its various components, thus the need for different tools from Weka. Our evaluation in this phase was done as follows:

First, from the "Human Rankings Collection" phase, for each judge and for each of his/her rankings, we extracted the identity of the ranked developer, the heuristics considered by the judge, and the judge's ranking of this developer.

Next, these data were used to design the input for both our proposed 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, the ranked developer's ID, and the values that represent the developer's expertise/sociality, depending on the scenario being worked on. Since we have 10 judges and each of them has ranked the top three experts in this scenario, we have 30 instances in total for each scenario. Table 6 shows the instance for Charles Chan whose ID is "16" and was determined by Judge-5 to be the second ranked expert.

Values	Social Heuristics within the DSSD
<instance>	
<value>J5</value>	Judge's ID
<value>D16</value>	Ranked Developer's ID
<value>10</value>	Number of shared files with the Active Developer
<value>23</value>	Number of shared commits with the Active Developer
<value>0</value>	Number of shared files within the DSSD
<value>0</value>	Number of shared commits within the DSSD
<value>2</value>	Ranking
</instance>	

Table 6: Scenario-2 Rankings (RS Algorithm and NaiveBayes Input Example)

After we created all the judge instances, we applied our algorithm to the heuristic values in these instances in order to generate the algorithm's ranking of the developers in the instances. Then, we compared for each instance the judge's ranking and our algorithm's ranking in order to evaluate its performance.

Instances	Judges	Ranked Developers	Judge Rankings	Naive-Bayes Rankings	Recommender System Algorithm Rankings
...					
13	5	Scott Hernandez	1	2	3
14		Charles Chan	2	3	2
15		Drieseng	3	1	1
...					

Table 7: Scenario-2 Judges, NaiveBayes, and RS Algorithm Rankings Comparisons

We also applied the NaiveBayes, NaiveNet, and J48 machine leaning 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, as shown in Table 7, in order to evaluate its performance.

In the following section, we show the results from our analysis for each scenario and groups as well. We will limit our explanation in this section to the rankings by NaiveBayes. Then, in the Discussion we will also look at the results of the NaiveNet and J48 machine learning algorithms.

4.2 RESULTS

In this section, we first represent the weights reflecting the importance and priority of the heuristics and groups. After that, we show how accurate our algorithm is in recommending the experts.

4.2.1 Heuristic and Group Weights

Figure 2 shows the weights of each technical heuristic. As mentioned in the "Identifying Experts" section, *Type-1*, *Type-2*, and *Type-3* clones are considered to be one heuristic, but we separated them in the analysis here 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. We found that the developers who worked on *Type-3* clones might have the best expertise since their changes to a code fragment show that they might have good understanding of the logic of that code. However, we see in the figure that *Type-1* and *Type-2* clones have higher weights, but these weights do not arise due to the importance of the types themselves but because some developers worked on both types. In our analyses of other technical heuristics using the overall type heuristic (i.e., ignoring the three subtypes), we found that judges think that the developers who have modified a large number of lines in the fragments that are clones of the current code might be the ones who have good expertise. Moreover, among those developers if some of them have modified code more 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 clone fragments might be enough to consider them as experts without considering which types they have cooperated on.

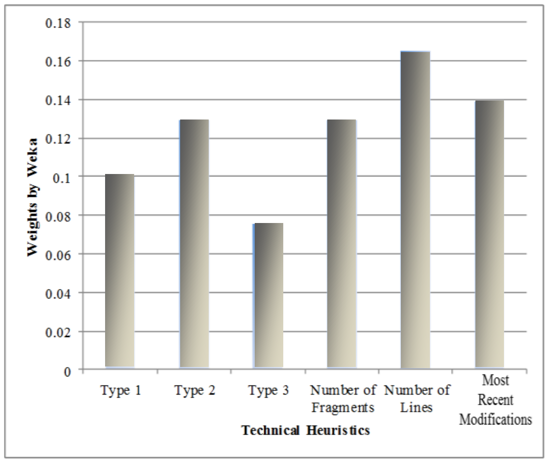

Figure 2: Technical Heuristics Weights

Regarding the four heuristics under sociality within the DSSD (see Figure 3 below), we see from the figure that the social relationship with the Active Developer is considered slightly more important to consider than the sociality within the DSSD. Judges, on the other hand, did not seem to pay attention to the basis of these relationships (counting number of shared files and commits) while they were ranking the experts.

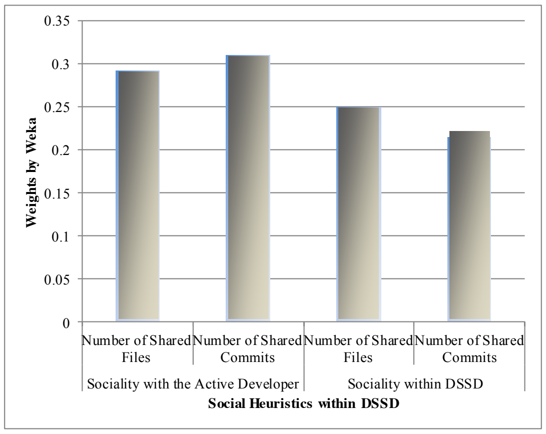

Figure 3: Social Heuristics Weights within DSSD

Figure 4 shows the weights of the heuristics for sociality within the ERSF. This figure shows that *Trust* in the relationship with the Active Developer is given higher weight, followed by *Response*, and then *Help* with the lowest weight. The reason behind this pattern is the dependency of *Help* on *Response* and the dependency of *Response* on *Trust* that caused the variation in weights between these three heuristics (and not their importance). The same thing happened with sociality within the ERSF. However, the high value for *Response* within the ERSF shows that the *Response* heuristic has higher importance than the others.

Regarding the low value of *Recommendation* with the Active Developer in Figure 4, its value is not dependent on any other heuristic; thus, to have the lowest weight means that it has the lowest importance compared to other heuristics with the Active Developer. On the other hand, *Recommendation* within the ERSF has similar weight to the *Helpfulness* heuristic, which means it has similar importance as the *Helpfulness* heuristic.

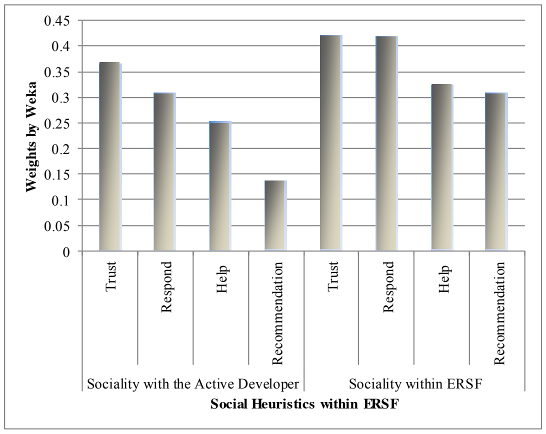

Figure 4: Social Heuristics Weights within ERSF

After we analyzed the importance of the heuristics compared to each other within each group, we also studied the importance of the groups themselves compared to each other. Figure 5(a) shows that human judges prefer to get help from socially capable developers more so than getting help from the developers who just have technical expertise. We also see in Figure 5(b) that the social heuristic within the DSSD lost its importance when the ERSF was used. This shows that judges prefer to get assistance from the developers who have demonstrated their ability to help more than getting help from the developers with whom they have worked within the DSSD but who have not demonstrated helping ability before.

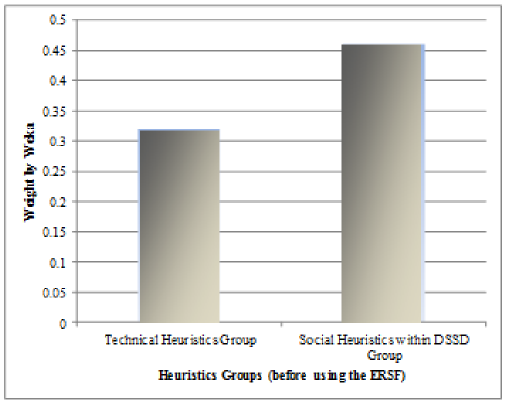

Figure 5-a: Group Weights (Before Using the Recommender System)

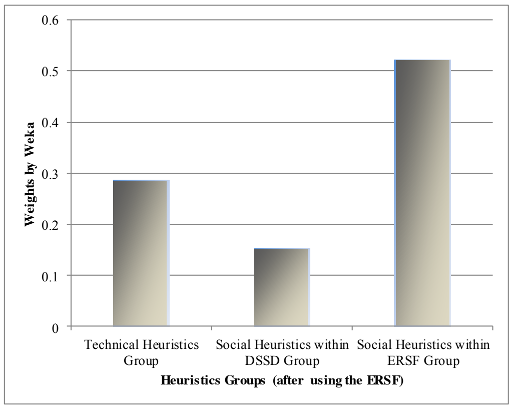

Figure 8-a: Group Weights (Before Using the Recommender System)

4.2.2 Algorithm Accuracy

A recommender system's performance is evaluated by the accuracy of its recommendations. We have done this by calculating the precision (representing the percentage of recommendations that are correct) and the recall (representing the percentage of correct experts recommended) of our recommender system algorithm against the human judges as well as NaiveBayes, J48, and NaiveNet machine learning algorithms in each group (technical heuristics, social heuristics within the DSSD, and social heuristics within the ERSF) and the groups themselves.

The precision and the recall measures are designed based on three measurements: (1) False Positive contains the experts recommended by our algorithm but not by the judges, (2) False Negative contains the experts recommended by the judges but not by our algorithm, and (3) True Positive contains the intersection of the algorithm recommendations and the rankings by the judges as of McDonald [10].

We first compared our algorithm rankings to the rankings by each judge using NaiveBayes in order to measure the True Positive of the comparison with this judge. 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.

The False Negative values for both the precision and the recall 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 (represented as "30" instances, as described) we also had "30" rankings by the machine learning algorithms. 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".

Table 8 shows the comparison between our algorithm rankings and the judge rankings as well as the comparison between our algorithm rankings and the NaiveBayes rankings. Our algorithm shows good to excellent precision and recall in its performance in Social Heuristics within the DSSD, Social Heuristics within ERSF, and the combined Groups ranking comparison as represented in Table 8. It also shows that the precision and recall values of the 3-Heuristics Groups Combination are greater than the values of the Technical Heuristics and the Sociality within the DSSD in both comparisons, against human judgments and machine learning algorithms. The technical heuristics do not come out so well, but could prove very useful for new developers whose sociality isn't known, thus helping overcome a cold start problem. But, the key result of this section is the importance of the social heuristics, both within the DSSD and the ERSF.

Metric / Comparison	Judge Rankings		NaiveBayes Rankings	
	Precision	Recall	Precision	Recall
Technical Heuristics	40%	40%	23%	23%
Social Heuristics within DSSD	63%	63%	60%	60%
Social Heuristics within ERSF	100%	100%	70%	70%
3-heuristics Groups Combination	70%	70%	63%	63%

Table 8: Precision and Recall

The NaiveNet and J48 algorithms show similar precision and recall values in the Social Heuristics within the DSSD, Social Heuristics within the ERSF, and the Group ranking comparison as NaiveBayes. However, they were different than NaiveBayes in the Technical Heuristics. First, for NaiveNet, the precision and the recall were 33%; second, for J48, the precision and the recall were 30% for the Technical Heuristics. Our explanation of these differences is the effect of the "Most recent modifications" heuristic since this is the only heuristic that explicitly uses temporal information, unlike other scenarios that were just concerned with heuristics that measured non-temporal characteristics.

Overall, the good to excellent precision and recall compared to both the human judges as well the NaiveBayes, NaiveNet, and J48 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 useful for organizations doing software system development to help their developers find suitable experts who can help in the code at hand.

5 Threats to the Validity

While the work presented in this paper is original and the proof of concept experiment shows that it is promising, there are some threats as well. First, to judge technical expertise, we exploit a clone detection tool, SimCad for detecting exact and near-miss clones of the code fragment the Active Developer is working on. Depending on the accuracy and thresholds of the tool, the recommendations might vary. However, SimCad has been shown to be a state of the art scalable clone detectors that detects both exact and near-miss clones with high accuracy [16]. Second, while we evaluated the approach with a proof of concept experiment, a real software development environment would be necessary to accurately and fully measure the performance of the system. However, it was impossible for us to test the system in a real environment, as the system needed to be running for a while in the development environment to get all the required historical data. We mitigated this threat by carefully designing three scenarios capturing different technical and social heuristics and then using ten users/judges and machine learning algorithms both to learn and then evaluate the approach. While more judges certainly would have allowed better evaluation of the system, the judges we did select had a software development background that allowed them to understand the three scenarios well. Thus, while preliminary, these results clearly show the promise of the proposed approach.

6 Conclusion

Our research is a proof of concept experiment demonstrating a recommender system that uses heuristics measuring both technical expertise and social ability to find appropriate developers to help software engineers over impasses as they develop code. The approach is innovative in using clones to find a much wider range of developers with appropriate technical expertise and that is helpful in overcoming cold start problems. It also uses a number of novel heuristics to measure social aspects of developers, including notions of trust, reputation, and helpfulness. The heuristics are computed based on data collected in the code repository, by the software development environment, and by the recommender system itself. This allows profiles of the developers to be automatically kept up to date over time as they use these sub-components, which in turn allows recommendations to be continuously adjusted to the evolving expertise of the developers.

We plan to provide our tool as an Eclipse plug-in to the public so that developers can use it to find experts to help them. This would be the ultimate test of the effectiveness of the system since real developers would be using it to get real help when confronted with actual impasses. The data collected in such a real world environment, over time (probably years!), would be invaluable in allowing the discovery of a much finer grained and genuine set of weights not subject to the artifice necessary in the experiments done to date. Nevertheless, we feel that the current weights would be a useful initial tuning for such a system that would make it much more effective than using random weights. Further details on this research can be found elsewhere [4].

Authors Biographies

Ghadeer Kintab was a M.Sc. student in Computer Science at the University of Saskatchewan. She is currently living in Jeddah – Saudi Arabia.

Chanchal Roy is Associate Professor of Computer Science at the U. of Saskatchewan. His expertise is in software engineering, particularly software maintenance and evolution.

Gord McCalla is Professor Emeritus of Computer Science at the U. of Saskatchewan. His expertise is in the areas of artificial intelligence in education, user modelling, and personalization.

References

[1] A. Begel, Y. P. Khoo, and T. Zimmerman, “Codebook: Discovering and exploiting relationships in software repositories,” *Proc. ICSE,* 2010, pp. 125–134.

[2] The DocShare Plugin: http://wiki.eclipse.org/DocShare_Plugin

[3] A. Guzzi, A. Begel, J. K. Miller, and K. Nareddy. “Facilitating enterprise software developer communication with CARES,” *Proc. ICSM,* 2012, pp. 527–536.

[4] G. Kintab, “Experts Recommender System Using Technical and Social Heuristics”, *M.Sc. Thesis, University of Saskatchewan,* 148 pp. 2013. Available at: http://goo.gl/960aOt

[5] 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,” *Trans. Softw. Eng.,* vol. 37, no. 3, pp. 307–324, 2011.

[6] B. Macek, M. Atzmueller, and G. Stumme, “Profile mining in CVS-logs and face-to-face contacts for recommending software developers,” *Proc. SocialCom/PASSAT,* 2011*,* pp. 250–257.

[7] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, “Expert recommendation with usage expertise,” *Proc. ICSM*, 2009, pp. 535–538.

[8] D. W. McDonald and M. S. Ackerman, “Just talk to me: A field study of expertise location,” *Proc. CSCW,* 1998, pp. 315–324.

[9] D. W. McDonald and M. S. Ackerman, “Expertise recommender: A flexible recommendation system and architecture,” *Proc. CSCW,* 2000, pp. 231–40.

[10] D. W. McDonald, “Evaluating expertise recommendations,” *Proc. CSCW*, 2001, pp. 214–23.

[11] S. Minto and G. C. Murphy, “Recommending emergent teams,” *Proc. MSR*, 2007, pp. 12–13.

[12] A. Mockus and J. D. Herbsleb, “Expertise browser: A quantitative approach to identifying expertise,” *Proc. ICSE*, 2002, pp. 503–512.

[13] A. Moraes, E. Silva, C. da Trindade, Y. Barbosa, and S. Meira, “Recommending experts using communication history,” *Proc*. ICSE, 2010, pp. 41–45.

[14] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Sci. Comput. Program.* 74(7): 470-495 (2009).

[15] D. Schuler and T. Zimmermann, “Mining usage expertise from version archives,” *Proc. MSR,* 2008, pp. 121–124.

[16] J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools", *Proc. ICSME*, 2014, 10 pp. (to appear).

[17] S. Uddin, and 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. WCRE*, 2011, pp. 13-22.

[18] I. H. Witten, E. Frank, and M. A. Hall, “Data Mining: Practical Machine Learning Tools and Techniques, Boston”, Elsevier, 2011.

[19] 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. RSSE,* 2008, 2pp.

[20] *Weka:* http://www.cs.waikato.ac.nz/ml/weka

[21] Y. Ye, Y. Yamamoto, and K. Nakakoji, “A socio-technical framework for supporting programmers,” *Proc. FSE, 2007*, pp. 351–360.