A Cost Effective Recommender System for Multi-Objective Test Case Prioritization

Maral Azizi
Departemant of Computer Sceince and Engineering
University of North Texas
Denton, TX, USA
maralazizi@my.unt.edu

Hyunsook Do
Departemant of Computer Sceince and Engineering
University of North Texas
Denton, TX, USA
hyunsook.do@unt.edu

***Abstract*—**

Keywords: Recommender systems, test case prioritization, regression testing

I. Introduction

Software systems undergo many changes during their lifetime, and often such changes can adversely affect the software. To avoid undesirable changes or unexpected bugs, software engineers need to test the overall functionality of the system before deploying a new release of the product. One of the common ways to evaluate system quality in a sequence of releases is regression testing. In regression testing, software engineers validate the software system to ensure that new changes have not introduced new faults or they dont affect the other parts of the system. However, modern software systems evolve frequently, and their size and complexity grow quickly, and thus the cost of regression testing can become too expensive [4]. To reduce the regression testing cost, many regression testing and maintenance approaches including test selection and test prioritization have been proposed [34].

To date, the majority of regression testing techniques have utilized various software metrics that are available from software repositories, such as the size and complexity of the application, code coverage, fault history information, and dependency relations among components. Further, various empirical studies have shown that the use of a certain metric or a combination of multiple metrics can improve the effectiveness of regression testing techniques better than other metrics. For example, Anderson et al. [5] empirically investigated the use of various code features mined from a large software repository, showing that these code features can help improve regression testing processes. However, we believe that, rather than simply picking one metric over another, adopting a recommender system that identifies more relevant metrics by considering software characteristics and the software testing environment would provide a better solution.

Recommender systems have been utilized to help reduce the decision making effort by providing a list of relevant items to users based on a users preferences or item attributes. For example, companies that produce daily-life applications, such as Netflix, Amazon, and many social networking applications [17] are adopting recommender systems to provide more personalized services so that they can attract more users. Recently, recommender systems have been used in software engineering areas to improve various software engineering tasks. For example, Mens et al. provide a source code recommendation system to help the developer by giving hints and suggestions or by correcting an existing code [33].

According to previous studies of recommender systems, a majority of proposed approaches focus on recommending the most relevant items without considering contextual information. However, contextual information such as time and cost can improve the efficiency of the recommended list by providing the most relevant items per their cost ratio. For example, in a typical test prioritization scenario, testers assume that all test cases and faults have same cost while, in practice, there is a wide range of test costs and fault impacts on the system that testers do not put into a context when providing a list of most important test cases. Moreover, there are variety of goals for test prioritization. For example, a tester's goal could be: increasing the code coverage, increasing the fault detection rate in a shorter time and increasing the system reliability. Thus, for a given prioritization goal, different prioritization techniques might be applicable. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than would a random order of test cases.

However, there are several research have been done in context-aware regression testing area but they have not been considered the other aspect of the regression causes while applying context information. For example, Do etl, [] provided improved cost-benefit models for use in assessing regression testing methodologies, that incorporate context and lifetime factors. In their study they provide a cost model based on contextual information without considering the other factors that can be the causes of regression issues such as change history. Zhange etl, [] proposed a time-aware test case prioritization using integer linear programming, but their work is limited to the code coverage information rather than test cost. Qu etl, [] used combinatorial interaction testing techniques to model configuration samples for use in regression testing and they proposed a normalized fault detection rate model, still in this study they did not consider the testing cost as a cost

factor.

Therefore, in this paper we present a recommender system that uses test execution time as test cost, code dependency and change history as a metrics for measuring the criticality portion of code that accounts for varying test case and fault costs. To further investigate these metrics and its applications, we present prioritization techniques that account for these varying costs, and results of a case study in which we apply these techniques under several different test cases and fault severity cost distributions.

In order to implement our recommender system, we used relevant information of test cases and the applications under test and we showed that using this information can improve the effectiveness of test prioritization technique when we consider test cost. We implemented a multi-objective test case prioritization technique by applying our recommender system and performed an empirical study using two open source software systems and one commercial system. The results of our study show that our proposed approach can improve the effectiveness of test case prioritization compared to four other control techniques. The main contributions of this research are as follows: (1) We propose a cost-effective recommender system that improves test prioritization and (2) we perform empirical evaluations of the proposed technique and four other control techniques.

The rest of the paper is organized as follows. In Section 2, we discuss the approach used in this research and formally define context aware recommender systems. Sections 3 and 4 present our empirical study, including the design, results, and analysis. Section 5 discusses the results and the implications of these results. Section 6 discusses potential threats to validity and Section 7 presents background and related work. Finally, in Section 8, we provide conclusions and discuss future work.

II. Approach

In this section, we describe our proposed approach which shown in Figure 1. As can be seen from Figure 1, our recommender system uses three dataset: 1) the change history, 2) the class dependency and 3) the test cases. There are two major activity in our proposed approach: Fault severity and cost aware test case prioritization. Below we describe each activity in detail.

A. Fault Severity Measurement

The goal of our recommender system is to find a sorted list of test cases to balance above mentioned datasets. To implement this approach we proposed a cost-effective recommender system that uses three datasets as input parameters for producing the list of most risky classes with considering their test cost. Traditionally, recommender systems use two main inputs for producing a ranking score for a specific item by the below formula:

$$R : User * Item$$

Those recommender systems show a significant effect and provide the most related list but still they ignore the importance of contextual information. Our recommender system takes fault severity and test cost as the input parameters for generating the list of most important test cases that should be tested first.

There are different ways to measure fault severity such as its fixing cost and the require time to find fault location, but here we define bug severity as a combination of the impact that a bug may induce on the system and the location of the bug. We defined our ranking formula as:

$$R : I_{Ci} * L_{Ci} * T_{Ci}$$

where I_{Ci} is the $changeimpactscore$, L_{Ci} is the score of location criticality and T_{Ci} is the test case cost for testing class C_i. In the following subsections, we explain the steps of calculating each parameter in detail.

1) Change History Analysis: The first phase of our proposed approach is change history analysis. Change history has a significant role in the error proneness of code, according to the previous studies []. Although other code attributes such as statistic code metrics also might have an impact on faulty code or failure risk, we decided to choose change history to avoid over-fitting. The process of change impact analysis has three major steps itself. First of all, we collected all change information and bug reports from the entire repository for each application. After collecting all change history information we applied feature selection technique to find the most effective metrics that are likely causes of system failure. To select the most effective features we measured gain ratio for all features. In next phase, to evaluate the impact that a particular change may impose to the system, we generated a linear model from a set of collected change metrics and fault history. We also evaluated the accuracy of our linear model.

In order to evaluate the accuracy of our classification model, we used 10-fold cross validation and we repeated this process for several times. Also, we used common accuracy indicator to determine the accuracy of our model. Below Table I shows the confusion matrix that we used for accuracy evaluation. PC indicates the percentage of correctly classified instances, TP indicates the number of classes that contains a bug and our classification model also classified them as a buggy classes, and FP is the number of classes that are classified as buggy but they are clean.

TABLE I: Confusion Matrix

Metrics	Predicted: No	Predicted: Yes	
Actual: No	TN = 33	FP = 63	96
Actual: Yes	FN = 54	TP = 46	100
	87	109	196

A higher PC value means that our classifier model is efficient. If PC is high, but the recall (TP) low, this means that our classifier classified a large number of classes as clean when they are not; this means that we missed many buggy classes during this phase for testing. However, if FP is high it

means that our model detected many files as buggy when they are clean and this will add extra cost for testing the system when it is not needed.

Finally, output of this part of our recommender system is I_{Ci}, which is a numerical value that indicates the impact of the change on a specific class.

2) Change Location Analysis: The second part of our recommender system is change location analyzer. The idea of analyzing change location is from the concept that some scopes of a program has a higher effect on system failure than other parts. Since our focus is to provide a cost effective model for test prioritization we consider the cost that a fault may induce on the system. For example, if a fault happens in a basic block the cost that it may impose to the system is much higher than if the same fault happens in a part of a system that is rarely used. Our hypothesis to evaluate the cost of a location is that classes with higher dependency are more costly compared to those with lower.

To measure the class dependency score, we extract class dependency from the source code of each subject application using the Understand tool[]. After, collecting all the class dependency from each application we compute the location criticality score by using the PageRank algorithm []. PageRank is an algorithm used by Google Search to rank websites in their search engine results. PageRank is a way of measuring the importance of website pages. According to Google: PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. PageRank algorithm equation is shown below.

$$PR(pi) = \frac{1-d}{N} + d \sum_{Pj \in M(Pi)} \frac{PR(pj)}{L(pj)}$$

where $C = \{p_1, p_2, ..., p_n\}$ are the pages under consideration, $M(p_i)$ is the set of pages that link to pi, $L_{(pj)}$ is the number of outbound links on page p_j, and N is the total number of pages.

We used this concept to rank our classes based on their dependency. Undrestand provides a list of classes with three dependency metrics: References, FromEnitity and ToEnity. References is the number of classes that referred to this class in total, FormEntity is a numerical value that indicates from how many classes this class received links and ToEntity is a numerical value that indicates to how many classes this particular class refer. The results of applying PageRank algorithm in our method is a numerical value L_{Ci} that indicates the relative importance of a class in the system from its number of input and output to other classes.

3) Test Case Cost: The third activity of our recommender system is test cost evaluation. Similarly to the fault severity, there is no general measure to determine test cost. Several measures could be considered when evaluating test cost such as hardware cost, type of a test like manual or automatic, test execution time, ect. Generally, there is a agreement that not all test cases have same costs []. We consider test execution time as a factor for measuring the cost value.

B. Test Case Prioritization

After collecting the three metrics explained above(change impact score, change location score and test cost), we calculate the final risk score using the following equation:

$$R_{Ci} : I_{Ci} * L_{Ci} * T_{Ci}$$

Where I_{Ci} is the $changeimpactscore$ of Ci, L_{Ci} is the change location score and T_{Ci} is the cost of test case that cover that particular component. Using R_{Ci} scores, our recommender system provides a ranked list of test case. The ranked

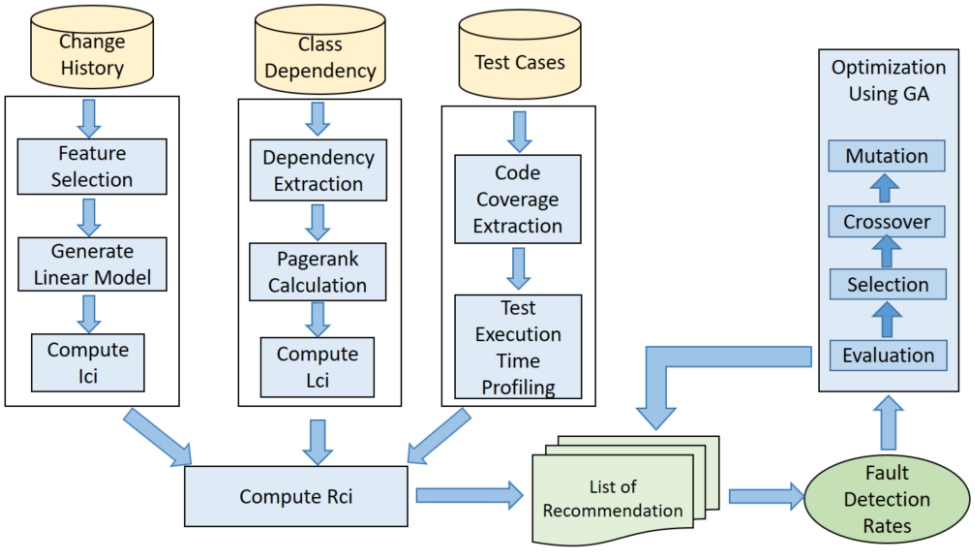

Fig. 1: An Overview of the Proposed Technique

list contains those classes of the system that are most likely to be the cause of regression faults and their testing cost is less expensive. Finally, we calculate the average percentage of fault detection scores for the applications.

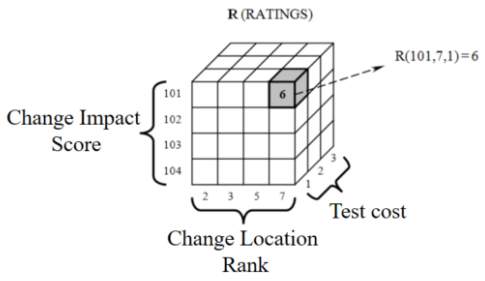

Fig. 2: An Example of Multidimensional Recommender System

Figure 2 shows an example of our recommender system. Value R in this figure indicates that, the ranking score for testing the class 101 of the system with location rank 7 and test cost 1 is equal to 6.

C. Optimization

The final phase of our recommender system include a optimization module which aimed for enhance the efficiency of the system. To do this, we applied *Genetic Algorithm* to suggest the best order of test case for optimum results for different testing goals. In this scenario we simulated different testing goals such as: test the system with time constraint, testing the system to get maximum fault detection percentage, and testing the most frequently used components of the system. Initial population of our GA are the output list of our recommender system.

1) Fitness Function: We defined fitness function as direct proportional to the average relevance of a recommendation to the test goal.

2) Crossover:

3) Mutation:

III. Empirical Study

As stated in Section I, the goal of this study is to assess our regression testing approach in terms of finding the ordered list of test cases that cover most risky parts of the system with considering the test cost. To investigate our research questions, we performed an empirical study.The following subsections present our objects of analysis, study setup, threats to validity, and data analysis. In particular, we address the following research questions:

RQ1: Are final scores effective at identifying high risky classes?
RQ2: Can our recommender system be useful for improving performance regression testing techniques?
RQ3: How effective is our proposed approach in terms of cost saving?

TABLE II: Subject Applications Properties

Metrics	DASCP	nopCommerce	Coevery
Classes	107	1,919	2,258
Files	201	1,473	1,875
Functions	940	21,057	13,041
LOC	35,122	226,354	120,897
Sessions	748	1310	274
Faults	35	70	30
Version	3	23	3
Test Cases	95	543	1,120
Installations	3	2	1

A. Objects of Analysis

We used three web applications to evaluate our proposed technique. Table II lists these applications. Our first object of analysis is DASCP, which is a digital archive and scan software for civil projects; we obtained this application from a private company. DASCP is a web based application designed to store civil project contracts, which includes the technical information of civil and construction projects such as project plans and relevant associated information. DASCP includes an access control system and provides two types of access rights: one user group has permission to edit or insert a projects information or upload maps and contract sheets and the other user group is only allowed to view the data and details about the projects.

Our second application is nopCommerce, which is a widelyused open source e-commerce shopping cart web application with more than 1.8 million downloads. This application is written in ASP.Net MVC and uses Microsoft SQL Server as a database system [1]. Our last application is Coevery, which is an open source customer relationship management (CRM) system written in ASP.Net. This application provides an easy framework for users to create their own customized modules without having to write any code. The UI design of Coevery has been developed by AngularJS and Orchard Technologies [2].

Table 1 lists the applications under study and their associated data: Classes (the number of class files), Files (the number of files), Functions (the number of functions/methods), LOC (the number of lines of code), Sessions (the number of user sessions that we collected), Faults (the number of seeded faults), Version (the number of versions), Test Cases (the number of test cases), and Installations (the number of different locations where the applications were installed).

B. Variables and Measures

1) Independent Variables: To investigate our research questions, we manipulated one independent variable: prioritization technique. We considered five different test case prioritization techniques, which we classified into two groups: control and heuristic. Table III summarizes these groups and techniques. The second column shows prioritization techniques for each group, and the third column is a short description of prioritization techniques.

TABLE III: Test Case Prioritization Techniques

Group	Technique	Description
Control	Change history-based (T_{ch})	Test case prioritization based on change impact analysis score
	Most frequent web forms-based (T_{mfw})	Test case prioritization based on value of frequency for each web form.
	Most frequent components-based (T_{mfc})	Test case prioritization based on value of frequency for each component.
	Random (T_r)	Test execution in random order.
Heuristic	Hybrid collaborative filtering-based (T_{hcf})	Test case prioritization based on the proposed technique.

As shown in Table III, we considered four control techniques and one heuristic technique. For our heuristic technique, we used the approach explained in Section II,

2) Dependent Variable and Measures: Our dependent variable RQ1 is the average percentage of fault detection (APFD) referring to the average percentage of faults detected during the test suite execution. The range of APFD is from 0 to 100, the higher value indicating better prioritization technique. Given T as a test suite with n test cases and m number of faults, F as a collection of detected faults by T and TF_i as the first test case that catches the fault i, we calculate APFD [**?**] as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

RQ2 seeks to measure the effectiveness of our proposed approach when we have constrained resources, which means that we need to evaluate the effectiveness of our approach using a different metric. Qu et al. [**?**] defined the normalized metric of APFD, which is the area under the curve when the numbers of test cases or faults are not consistent. The NAPFD formula is as follows:

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n}(t_j - \frac{1}{2}t_{TF_i})))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i}$$

In this formula, n is percentage of the test suite run, m represents the total number of faults found by all test cases, TF_i indicates the same parameter as AFPF, and p is the number of faults detected by percentage of our budget divided by total number of detected faults when running 100% of test cases.

C. Experiment Setup

D. Data Collection

To perform our experiment we collected different types of data from our AUTs. Our first dataset is change history. The second one is class dependency and the last one is test case code coverage information. We explain the data collection processes in the following subsections.

1) Collect Code Change History: We had to take three steps to measure change impact. First, we needed a clear understanding of the applications with respect to their changes. For instance, we needed to check whether a change was just renaming a variable or component, the addition of some comments, or an alternation of code by adding or deleting functions. Then, we needed to check whether changes had been made in the current version, and finally, we tested a recently changed system [**?**]. In order to collect change history information for training, we used all versions of our applications.

In our study, we collected the change history of our three applications. We chose ten metrics that have high correlations with bugs. Most of these metrics have been used in bug detection research, and they are known to be good indicators for locating bugs [**?**], [**?**], [**?**], [**?**], [**?**]. Table IV shows the applied change metrics in this study.

TABLE IV: Change metrics used to evaluate risk in this study

Metrics Name	**Description**
Revision	Number of revision of a component
LOC Added	Added lines of code
Max LOC Added	Maximum added lines of code
AVE LOC Added	Average added lines of code
LOC Deleted	Deleted lines of code
Max LOC Deleted	Maximum deleted line
AVE LOC Deleted	Average deleted lines of code
Code Churn	Sum of change in all revisions
Age	Age of the component in days from last release
Time	Time of the change in dd-mm-yyyy format

2) Collect Class Dependency: To extract class dependency information we used the Understand tool []. The Understand tool extracts 24 file-level and 18 class-level metrics such as Chidamber and Kemerer [5] and Object-Oriented metrics. If a file has more than one class, we derived file-level metrics from multiple class-level metrics. The Understand tool mostly provides two kinds of metrics: Avg* and Count*. To generate file-level metrics from multiple classes in a file, we averaged Avg* class-level metrics. However, when we get file-level metrics from Count* classs-level metrics, we added the values together. We used all 42 metrics for our experiments. The output of Understand is a list of classes with their numerical value of number of its references, from and to entities.

3) Collect Code Coverage and Test Cost: We collect code coverage data for test cases using the Visual Studio Test Analyzer tool. Also we used the same tool to collect the test case execution time. After collecting all required information, we created a SQL database and we stored them into a datatable. Our datatable records information about which tests exercised each class and the time its needed to be executed. We assigned a unique identifier to each class to provide a key for classes, which makes the mapping process much easier. Table **??** shows a schema of our datatable.

E. Fault Injection

We evaluated our approach by running it against tests with and without regressions issues. To have tests with and without regressions, we injected source code level faults that are similar to the developer mistakes. To do this, we asked volunteer students to seed faults randomly into the system.

F. Data and Analysis

IV. Threats to Validity

The primary threat to validity of this study is the amount of user session data and the type of users who participated in this study. For the private application that we used in this study, we collected user interaction data for a long period time; the collected data was created by actual users of the application. However, for the two open source applications, the period of time that we collected user interactions was relatively short, and the participants were not domain experts or regular users of the applications so their usage patterns had wide variations. This threat can be addressed by performing additional studies that monitor user interactions over a longer time period among a wider population, by considering industrial applications and different types of applications (e.g., mobile applications).

Another threat to validity is the choice of algorithms that classify components' frequency access ranking and analyze change impact. In this study, we applied various algorithms to create our classification model, but many other classification algorithms (e.g., decision tree and apriori algorithms) are available, and they could produce different results. The results can vary depending on the type of classification algorithms, the parameters set for classification algorithms, the variables being analyzed, and the environmental settings.

There is another concern regarding the bug reports that we used. Our classification prediction values for designing linear models in the change impact analysis were generated from bug history that was reported by actual users. Further, using these bug reports, we measured the coefficient of other variables to create our linear model for change impact analysis. Because our bug report data is not comprehensive and contains only those bugs accrued until the time that we stopped collecting data, and because there might be other bugs that have not been reported yet or that might occur later, there is a possibility that the bug reports are biased.

V. Related Work

In this section, we discuss studies of regression testing focusing on test case prioritization techniques that are most closely related to our work. Further, we discuss recommender systems and research related to that topic.

Test Case Prioritization: Test case prioritization techniques reorder test cases to maximize some objective functions, such as detecting defects as early as possible. Due to the appealing benefits of test case prioritization in practice, many researchers and practitioners have proposed and studied various test case prioritization techniques. For example, these techniques help engineers discover faults early in testing, which allows them to begin debugging earlier. In this case, entire test suites may still be executed, which avoids the potential drawbacks associated with omitting test cases. Recent surveys [**?**], [**?**] provide a comprehensive understanding of overall trends of the techniques and suggest areas for improvement.

Depending on the types of information available, various test case prioritization techniques can be utilized, but the majority of prioritization techniques have used source code information to implement the techniques. For instance, many researchers have utilized code coverage information to implement prioritization techniques [1], [2], [3]. Although this approach is simple and naïve, many empirical studies have shown that this approach can be effective [**?**], [**?**], [**?**], [**?**]. Recent prioritization techniques have used other types of code information, such as slices [**?**], change history [**?**], code modification information, and fault proneness of code [**?**].

More recently, several prioritization techniques utilizing other types of information have also been proposed. For example, Anderson et al. applied telemetry data to compute fingerprints to extract usage patterns and for test prioritization [**?**]. Memon and Amalfitano performed a study in which they applied telemetry data to generate usage pattern profiles [**?**]. In another study, Amalfitano et al. built finite state models based on usage data that they collected from rich Internet applications [**?**]. Carlson et al. [**?**] presented clustering-based techniques that utilize real fault history information including code coverage. Anderson et al. [**?**] investigated the use of various code features mined from a large software repository to improve regression testing techniques. Gethers et al. presented a method that uses textual change of source code to estimate an impact set [**?**].

Context-aware Recommender system: Recommender systems are software engineering tools that make the decision making process easier by providing a list of relevant items. There are three primary categories in recommender systems: content-based algorithms, collaborative filtering algorithms, and hybrid approaches [**?**]. Recommender systems are commonly used by users in their daily routines, helping in such tasks as finding their target items more easily. Some widely-used applications that provide recommender systems are Amazon, Facebook, and Netflix. These applications provide suggestions to target users based on the user or item characteristic similarities.

Further, in the area of software engineering, due to the decline in hardware facility prices, a variety of information is collected by software providers, such as change history, issue reports and databases, user log files, and so on. With the fast growth of such information, machine learning technologies motivate software engineers to apply recommendation systems in software development. Recommender systems in software engineering have been applied to improve software quality and to address the challenges of development process [**?**]. For instance, Murakami et al. [**?**] proposed a technique that uses user editing activities detecting code relevant to existing methods. Christidis et al. [**?**] implemented a recommender system to display developer activities by using information artifacts with quantitative metrics. Danylenko and Lowe pro-

vided a context-aware recommender system to automate a decision-making process for determining the efficiency of non-functional requirements [**?**].

As we discussed briefly, there are many types of information available for implementing test case prioritization techniques. In this research, we collected over 2,000 user sessions from three different web applications and gathered the change history of each application. Our research seeks to apply item-based collaborative filtering algorithms to generate a recommendation list of test cases for test prioritization. To our knowledge, our recommender system-based prioritization technique is novel and has not yet been explored in the regression testing area.

VI. Conclusions and Future Work

In this research, we proposed a new recommender system to improve the effectiveness of test case prioritization. Our recommender system uses three datasets (code coverage, change history, and user sessions) to produce a list of most risky components of a system for regression testing. We applied our recommender system using two open source applications and one industrial application to investigate whether our approach can be effective compared to four different control techniques. The results of our study indicate that our recommender system can help improve test prioritization; also, the performance of our approach was particularly noteworthy outstanding when we had a limited time budget.

Because our initial attempt to use recommender systems in the area of regression testing showed promising results, we plan to investigate this approach further by considering various algorithms (e.g., a context-aware algorithm and a hybrid algorithm), the characteristics of applications, and the testing context. For example, as we discussed earlier, there are some limitations in collaborative filtering. In future research, we plan to investigate other approaches to address a sparsity problem by applying an associative retrieval framework and related spreading activation algorithms to track user transitive interactions through their previous interactions. Also, to address the "New Item Problem," we plan to apply knowledge-based techniques such as case-based reasoning. Further, in this study, we did not evaluate our approach considering cost-benefit tradeoffs, so we would like to investigate this aspect in a future study. Also, we wish to apply our recommender system to different software domains such as mobile applications and perform additional studies that monitor user interactions over a longer period of time.

Acknowledgments

References

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[2] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[3] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.