A Collaborative Filtering Recommender System for Test Case Prioritization in Web Applications

Author 1
Departemant of ABC
University of x
X, X, X
X@ABC.edu

Author 2
Departemant of ABC
University of x
X, X, X
X@ABC.edu

***Abstract*—The use of more relevant metrics of software systems could improve various software engineering tasks, but identifying relationships among metrics is not simple and can be very time consuming. Recommender systems can help with this decision-making process; many applications have utilized these systems to improve the performance of their applications. To investigate the potential benefits of recommender systems in regression testing, we implemented an item-based collaborative filtering recommender system that uses user interaction data and application change history information to develop a test case prioritization technique. To evaluate our approach, we performed an empirical study using three applications with multiple versions by comparing four control techniques. Our results indicate that our recommender system can help improve the effectiveness of test prioritization; the performance of our approach was particularly noteworthy when we were under a time constraint.**

Keywords: Recommender system, test case prioritization, regression testing, risk measurement, code quality.

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 that they don't 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 regression testing cost, many regression testing and maintenance approaches including test selection and test prioritization [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 user's preference 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 hint and suggestions or by correcting an existing code [33]. Zanjani et al. performed a study by developing a recommender system for code review based on historical contributions of prior reviews [35]. Anvik et al. conducted research that applied machine learning techniques to developers and bug history to make suggestions about "who should fix this bug?" [6]. While many software engineering techniques have started to incorporate recommendation systems, no researchers have investigated the use of recommender systems in the area of regression testing.

Therefore, we investigate whether the use of recommender systems can help improve regression testing techniques, in particular focusing on test case prioritization. In this paper, we propose an item-based collaborative filtering recommender system that uses the user interactions data and application change history information. We implemented a 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 recommender system approach can improve the effectiveness of test case prioritization compared to four other control techniques. Further, our results indicate that our proposed approach yielded higher scores than traditional prioritization techniques when companies have limited time budgets.

The main contributions of this research are as follows: (1) We proposed an item-based recommender system that improves test prioritization, (2) We performed empirical evaluations of the proposed technique and four other control techniques, and (3) We provided guidance to testers and discussed practical considerations when they apply recommendation systems during testing.

The rest of the paper is organized as follows. In Section II, we discuss the approach used in this research and formally define collaborative filtering recommender systems. Sections III and IV present our empirical study, including design, results, and analysis. Section V discusses the results and the implications of these results. Section VI discusses potential threats to validity and Section VII presents background and related work. Finally, in Section VIII, we provide conclusions and discuss future work.

II. The Proposed Approach

In this section, we describe the proposed approach, which utilizes an item-based recommender system to improve test case prioritization techniques.

Figure 1 shows an overview of our proposed technique. There are three major activities in our approach: 1) usage pattern extraction, 2) change impact analysis, and 3) test case prioritization. The first two activities are used to produce a set of recommendations, which is utilized to prioritize test cases.

Before we describe these activities in detail, following is a brief overview of our approach and how these activities are related to each other. Note that in our approach we defined the methods as the components.

- Usage Pattern Extraction. In this step, our goal is to calculate the frequency of usage of each component. The upper box in Figure 1 shows this process. This step contains three major parts: user session data collection, components similarity calculation and usage rank prediction. The output of this step is a matrix of frequency scores for the components in method granularity.
- Change Impact Analysis. In this step, we are looking for a linear model to evaluate the impact of change on the system. In order to measure change impact, we collect the change history of the applications. We also obtain a matrix of risk scores for components. The lower box in Figure 1 shows this phase.
- Test Case Prioritization. After obtaining necessary metrics of frequency and change impact scores to produce a set of recommendations, we reorder test cases based on these recommendations.

In the following subsections, we will explain in detail how our recommender system works and how we apply this technique to test case prioritization.

A. Usage Pattern Extraction

The main focus of an item-based collaborative filtering recommender system is the user's rating of existing components. Our proposed technique uses two attributes: the most frequently used components and change history.

The goal of our hybrid recommender system is to suggest the highest risky components with the most access frequency among all other components in the applications. In large scale applications there are wide range of features and components. However, in reality users are not using all the functionality of the entire system. More often, a relatively small subset of entire system is being used by users. Therefore, even if a bug

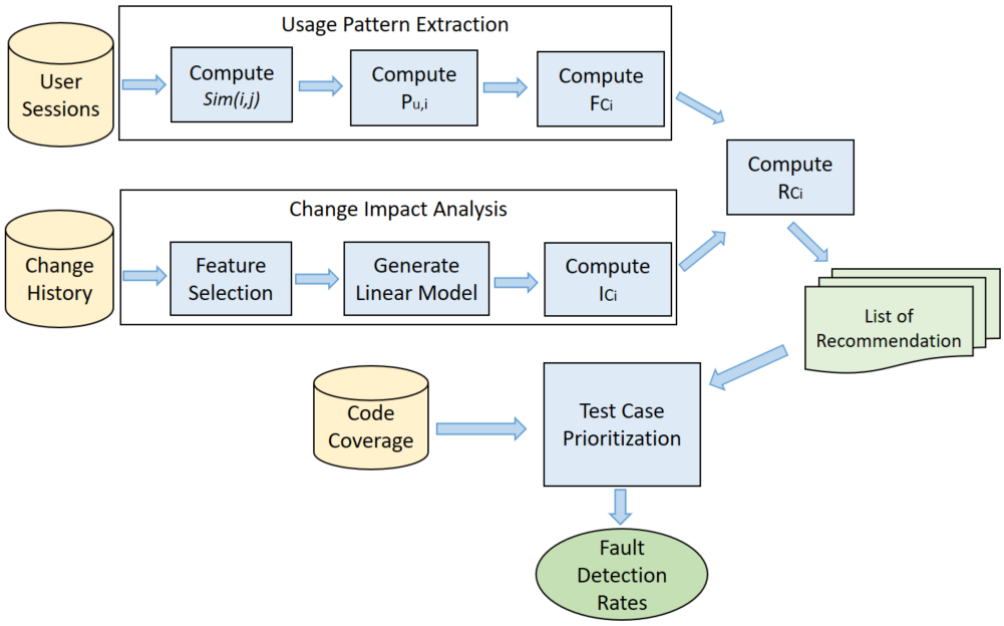

Fig. 1: An Overview of Our Test Case Recommender System

exist in a part of a application that no user ever is impacted by it, that bug has less impact in user-perceived reliability than more frequently access components. Our hypothesis is that by prioritizing test cases that exercise more frequently accessed components, we can obtain a higher rate of fault detection of the ordered test cases.

To do this, we use two collected datasets: a list of users $U = \{u_1, u_2, ..., u_m\}$ and a list of our components $C = \{c_1, c_2, ..., c_n\}$. Each user, u_i, has a list of components, c_{ui}, that the user has utilized in performing at least one task on c. Typically, in recommender systems, prediction is based on the numerical values of ratings from active users, but in our case we do not have access to such rating modules; instead, we consider the value of frequency access for a specific component by an individual active user as a rating score.

There are two primary techniques for collaborative filtering algorithms: user-based and item-based algorithms. In user-based collaboration filtering, we seek to find those users who are most similar to the current active user. In the item-based algorithm, first we design a model of user rating, and then we evaluate the expected ratings of an item based on the previous rating of the other similar items.

Suppose that the target user has rated a set of components, $\{c_1, c_2, ..., c_n\}$. The item-based collaborative algorithm computes the similarity between components by comparing the weighted average of the target user's ratings on these similar sets of components. In the next phase, the algorithm selects the k most similar components $\{c_1, c_2, ..., c_k\}$.

Further, corresponding similarities $\{s_{i1}, s_{i2}, ..., s_{ik}\}$ between components are also estimated. After finding the components that are most similar, we need to predict the ratings of the rest of the components that have been ignored or have not been used yet. To do this, the algorithm uses a weighted average of the target user ratings on these similar components. Below, we describe the process of computing similarity and ranking.

Figure 2 illustrates an example of component similarity prediction. The left-hand matrix in the picture shows how many times each component has been used by a user, and the right-hand matrix shows the predicted similarity values. Each number in the left matrix shows the frequency accessed by the ith user on the component C_i. The numbers are in the numerical scales, and NA indicates that the user has not used that particular component yet. To generate a list of $Top-N$ components, we need to calculate the similarity between user usage patterns and components. Next, we explain the process of evaluating $Top-N$ components.

In order to determine the similarity between two components i and j, we used Pearson-r correlation. If U is the set of users who rated components i and j, then we compute the correlation similarity using the following equation:

$$Sim(i,j) = \frac{\sum_{u \in U}(C_{u,i} - \bar{C_i})(C_{u,j} - \bar{C_j})}{\sqrt{\sum_{u \in U}(C_{u,i} - \bar{C_i})^2}\sqrt{\sum_{u \in U}(C_{u,j} - \bar{C_j})^2}}$$

where $C_{u,i}$ is the value of frequency access for user u on the component i, and $\bar{C_i}$ is the average value of frequency for the $i-th$ component.

To estimate the frequency rates of ignored components, we performed ratio prediction computation using a weighted sum method. This method provides the ratio prediction of a specific component i for user u based on similar components.

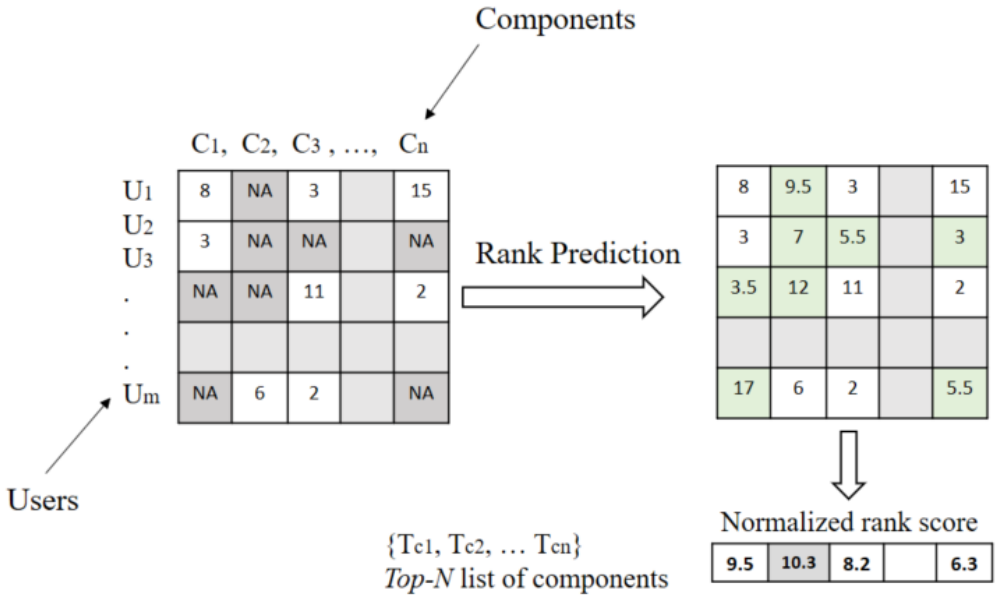

Fig. 2: Collaborative Filtering Process for Identifying Most Frequent Components

$$P_{u,i} = \frac{\sum_{all\ simillar\ components,N}(S_{i,N} * R_{u,N})}{\sum_{all\ simillar\ components,N}(S_{i,j})}$$

In this equation, $R_{u,N}$ is the rating score for user u and component N, and $S_{i,N}$ is the similarity score of component i and N. Once we calculate ratio scores for components, we can produce a matrix of components and their frequency ranking scores; now we can predict the frequency access ratio for those components that have not been used yet. In this step, we calculate normalized frequency scores for components using the following equation:

$$F_{Ci} = \frac{\sum(P_{Ci})}{number\ of\ users}$$

where P_{Ci} is the predicted rank score and F_{Ci} is normalized rank score of component i.

B. Change Impact Analysis

The second phase of our proposed approach is change impact analysis. Among hundreds of attributes of code and change history metrics to evaluate the code quality and proneness to error, we chose change history to identify the most risky components. A previous study [24] indicates that the use of change history information can be effective in detecting bugs. The details about change impact analysis are discussed in Section III-C. After completing the feature selection phase, we designed a linear model to analyze change impact. The output of our linear model is a matrix of components with their change impact values, I_{Ci} (the change impact score of component i).

C. Test Prioritization Using the Recommender System

After collecting the two metrics explained above (component risk scores and frequency ratios), we calculate the final risk score using the following equation:

$$R_{Ci} = F_{Ci} * I_{Ci}$$

Where F_{Ci} is the frequency score of component i and I_{Ci} is the change impact score of C_i.

Using R_{Ci} scores, our recommender system provides a ranked list of components. The ranked list of components contains those components of the system that are most likely to be the cause of regression faults. As shown in Figure 2, the test case prioritization algorithm reads two inputs (a recommended $Top-N$ list of components and code coverage of tests), and reorders test cases.

Finally, we calculate the average percentage of fault detection scores for the applications.

TABLE I: 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

III. Study

In this study, we investigate whether the use of a recommender system can improve the effectiveness of test case prioritization techniques. We consider the following research questions.

RQ1: Can our recommender system be useful for improving the effectiveness of test case prioritization techniques?

RQ2: Can we improve the fault detection rate when we have only limited budget of time to test the entire system?

The following subsections present our objects of analysis, study setup, threats to validity, and data analysis.

A. Objects of Analysis

To investigate our research questions, we performed an empirical study using two open source applications and one commercial web application.

DASCP 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 include 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 project's information or upload maps and contract sheets. The other user group is only allowed to view the data and details about the projects.

Our second application is **nopCommerce**, which is a widely-used 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 I 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).

Test cases were in application package and we did not implement any new test case. Version of open source applications were downloaded form the applications' *GitHub* repository and we downloaded all available versions.

B. Variables and Measures

1) Independent Variable: 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 II 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.

As shown in Table II, we considered four control techniques and one heuristic technique. For our heuristic technique, we used the approach explained in Section II, so, here, we only explain the control techniques we applied as follows:

Change History-Based (T_{ch}): The first control technique is test case prioritization based on change impact score. In order to perform this technique, we used the information that we obtained from the change impact analysis approach, which we explained in Section II-B. We prioritized our test cases based on the highest scores of the change impact matrix.

Most Frequent Web Forms-Based (T_{mfw}): This approach determines the web forms that have been most frequently used by users. We assume that the most frequently used web forms play a more important role in the system; eventually, they should have a higher priority to be tested first. Another reason why the frequency of web pages is the key for testing is that the most frequently used web pages usually contain more functionality and links with other pages, so any bugs on those pages can affect greater portion of the entire system. Here we define "base session" as a long session conducted by our users that displays the most interactions in the application. We picked 20% of our total sessions as base sessions. For example, in online shopping, our base session is a sequence of actions from user login to checkout, including all necessary actions and some random unnecessary actions such as browsing for other items, checking the inbox during the shopping process, etc.

After collecting all sessions, we conducted an analysis of web pages frequency by comparing the observed web forms in each session with a base session.

$$F_{w,i} = \frac{\sum_{page\ score\ for\ each\ session}(S_i)}{\sum_{number\ of\ base\ sessions}(BS_i)}$$

Using Figure 3 as an example, the page "PublicStore" from the base session was observed in test sessions 2 and 3. If we assume that we have ten test sessions and three base sessions, and this particular page was observed in seven of the ten pages, then the frequency of page "PublicStore" is equal to $(0.7/3) = 0.23$

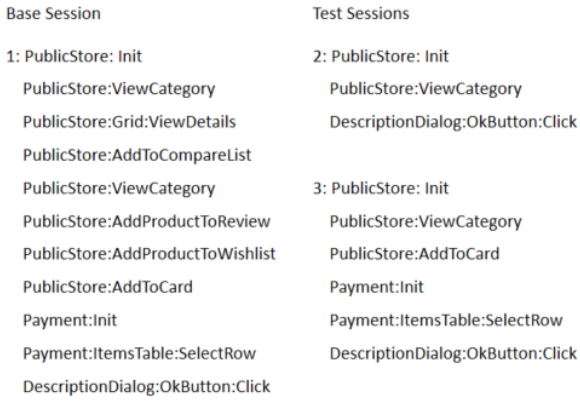

Fig. 3: Sample of Base Session and Test Sessions

Most Frequent Methods-Based (T_{mfm}): The most frequent methods approach is nearly identical to the web form frequency technique. The only difference is that we considered a method instead of a web form as a comparison factor. The most frequent methods, usually have high dependency on the other classes and methods. If one of them fails, it can cause a significant failure or degradation of the system. In order to prevent a domino effect in the system, high frequency methods should be tested first, because their failure can cause other components failure due to their dependencies.

Random (T_r): Random prioritization selects a test cases in random order. In this control technique, we randomly selected test cases until we had executed 100% of the test cases.

2) Dependent Variable: 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 [29] 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. [27] 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:

$$NAPFD = p - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{p}{2n}$$

In this formula, n is percentage of the test suite run, m represents the total number of faults found by all test cases,

TABLE II: 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 methods-based (T_{mfm})	Test case prioritization based on value of frequency for each method.
	Random (T_r)	Test execution in random order.
	Greedy (T_g)	Test case prioritization in based on code coverage.
Heuristic	Hybrid collaborative filtering-based (T_{hcf})	Test case prioritization based on the proposed technique.
	Reverse Hybrid collaborative filtering-based (T_{hcfr})	Test case prioritization based on the proposed technique with descending ranking values.

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. Data Collection and Experimental Setup

In order to perform our experiment, for both the control and heuristic techniques we needed to collect three different types of datasets: telemetry data, change history, and code coverage information. We explain the data collection processes in the following subsections.

1) Collection of Telemetry Data: To collect telemetry data, we implemented a small function to record user interactions. We considered a sequence of each user's interactions on a specific date as a user session. First, we uploaded two applications, *Coevery* and *nopCommerce*, on IIS server at the University of ABC in November 2016. The server specification is CPU Core i7 with 16 GB of RAM. After deploying our applications, we recruited volunteer graduate and undergraduate computer science students and assigned a variety of tasks to them. Assigned task to the volunteers were simple scenarios that each application is designed for. For example, in *nopCommerce* we asked them to perform online shopping by taking the actual steps which starts from login to payment. We also asked some of the users to be the system administrator so we could monitor the whole system rather than only the end user side. We also asked the end users to check the other part of the system randomly like checking their inbox or wish-list. In total, seventy volunteer students performed different tasks during a 40 days period. In total we collected 1310 and 274 user sessions for *nopCommerce* and *Coevery* respectively.

The data collection process for *DASCP* is different from that of *Coevery* and *nopCommerce*. *DASCP* is a commercial and closed source application that has three versions, and it has been installed on the servers of three companies since 2011. The DASCP users whose data we examined are real users, and they have application domain knowledge. We collected twelve-months period of user interactions for DASCP. As described in Section III-A, DASCP provides two types of access rights for users. In this study, we only considered the sessions of those users who have full access to the system. In total, 748 user sessions were collected during that period of time.

However, the length of the sessions varied by user, date and and workplace. For example, some users performed all their assigned task few hours before the determined deadline, while others distributed their tasks into several days. Average length of user session for *nopCommerce* is equal to 56 and for *Coevery* is 24. Although, in some case we obtained sessions length over 300, specially when the interaction date was close to the deadline. Also, most of the *Coevery* users are selected from graduate students since the functionality of this application is relatively more complex than *nopCommerce*.

Figure 4 shows an example of the raw telemetry data. The left column shows the session identifier, which is a user navigating through the system. The right column is the set of server side user interactions. The structure of the interactions is of the format (Form name):(Control name):(Action name).

0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:PublicStore:Init
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:ViewCategory:Init
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:ItemsGrid:ViewDetails
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:AddProductToReview:Click
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:AddProductToWishlist:Click
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:AddToCard:Click
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	PublicStore:Payment:Init
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	Payment:ItemsTable:SelectRow
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	Payment:DescriptionDialog:Click
0E98L725-M51C-4BFA-9960-E1C80M27ABA0	Payment:OkButton:Click

Fig. 4: Sample User Session

2) Collection of 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 the renaming a variable or component, the addition of some comments, or an alternation of code by adding or deleting functions, and so on. Then, we needed to check whether changes had been made in the current version, and finally, we tested a recently changed system [7]. 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 [22], [32], [24], [20], [26]. Table III shows the applied change metrics in this study.

3) Collect Code Coverage: Once our recommender system was designed and implemented, we needed to find test cases that covered the recommended components. We collected the code coverage data for our test cases using code coverage analysis tool that Microsoft Visual Studio provides as part of its framework. After collecting the code coverage information, we entered that information into a relational database. We assigned unique identifier values for each method and test case

TABLE III: 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
Max Code Churn	Maximum code churn for all revisions
AVE Code Churn	Average code churn per revisions
Age	Age of a component in days from last release
Time	Time of a change in dd-mm-yyyy format

which provides a key for method and test tables. So we could easily map the methods to the test cases that exercise them.

TABLE IV: Code Coverage Data Table

MethodID	Risk Score	TestID1	...	TestID n
12	0.876	0		0
287	0.012	1		0
301	0.547	0		0
148	0.145	0		1
67	0.055	1		0

Table IV show the code coverage data we collected. The first column, "MethodID", shows the unique identifier values that we assigned for each method. The second column shows the final risk scores, which is the output of our recommender system. Other columns list our test cases with Boolean values: 0 indicates that the test case does not cover the method, and 1 indicates that the test case covers the method.

4) Seeded Faults: As mentioned in **??**, faults were seeded manually by graduate students. We tried to simulate the naturally developer faults into the applications. All seeded faults are in server-side code level and we ignored HTML-based and GUI faults. Four types of faults were seeded into the applications. First type is data faults, which are faults that related to the interacting with the data store. Logic faults that are logic error in code, action faults that modifies parameter values and actions, and finally, linkage faults that change the hyperlinks references.

Once the change history data is collected, we applied ANOVA analysis to our dataset to obtain a linear model. Our goal was to find the correlation coefficient for each metric to measure statistical relationships between a variable and real defects. The value of this measure fluctuates between 1 and -1, where 1 indicates a strong positive relationship, 0 indicates no correlation, and negative value means reverse correlation. For example, if we have a -0.87 correlation coefficient score for Age metric that indicates that the oldest components of the system has a less risk of regression faults. We also calculated root mean squared error and mean absolute error. A lower error value shows that the model has higher prediction accuracy.

In order to evaluate our linear model, we applied 10-fold cross validation. By applying linear regression, we obtained a model that determines the weight of each variable. In next step we applied our obtained model to evaluate the risk score of each component. Finally, by having a matrix of components and their relevant risk scores, we permuted the test cases. For example, suppose we have five components $C = \{c_1, c_2, ..., c_5\}$ with risk score as follows : $R = \{0.0014, 0.251, 0.034, 0.561, 0.138\}$. Also, suppose we have a list of test cases with their code coverage information. Then, we will reorder the test cases in such a way to test c_4 first, since it has highest risk score (0.561), and c_1 will be the last component to be tested.

After collecting all the required data, we ran control and heuristic techniques and calculated APFD and NAPFD values for the reordered test cases to determine whether the proposed technique improved the fault detection rate.

IV. Data and Analysis

In this section, we present the results of this study considering each research question.

A. RQ1 Analysis

RQ1 investigates whether the use of the recommender system can help improve the effectiveness of test case prioritization techniques. Figure 5 shows the results combined for all applications. The vertical axis shows APFD scores, and the horizontal axis shows prioritization techniques: "T_{ch}" (change impact-based), "T_{mfw}" (frequent web form-based), "T_{mfm}" (frequent component-based), "T_r" (random order) and "T_{hcf}" (recommender system-based).

As shown in Figure 5, the proposed technique (T_{hcf}) yielded higher APFD scores than all control techniques while its data distribution is slightly wider than others. In particular, T_{hcf} outperformed T_r by 56% in APFD score (the median APFD values for T_{hcf} and T_r are 80.59 and 59.39, respectively).

The first three control techniques produced very similar results with marginal differences. To understand how these techniques performed for each individual application, we constructed boxplots for each application.

Figure 6 presents the results for DASCP. Similar to the results that considered all applications, the heuristic (T_{hcf}) outperformed all control techniques, but data distribution patterns are different. The variances are small, but there are more outliers. Among control techniques, the random technique was the worst performer, whereas the frequent component-based technique (T_{mfm}) produced results similar to the change history based technique (T_{ch}) .

Figure 7 presents the results for nopCommerce. The results show that our proposed technique outperformed all four control techniques, but the variance of the APFD scores is relatively high. Among the first three control techniques, the change history-based technique (T_{ch}) produced slightly better results than the others. This application has 23 versions, and we had a sufficient amount of change history data for our

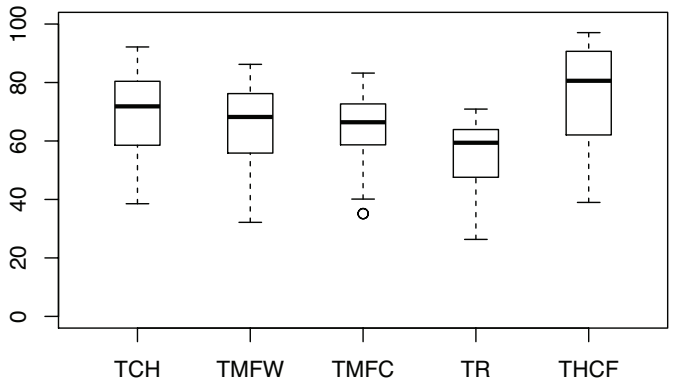

Fig. 5: APFD Values for All Applications

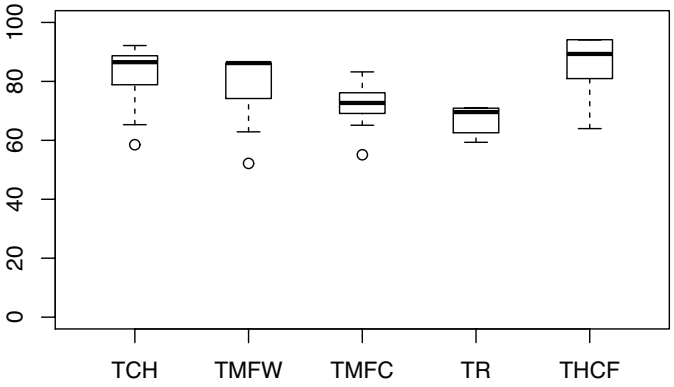

Fig. 6: DASCP - APFD Boxplots

training, so we speculate that this is the reason why the change history-based technique produced better results than the two control techniques.

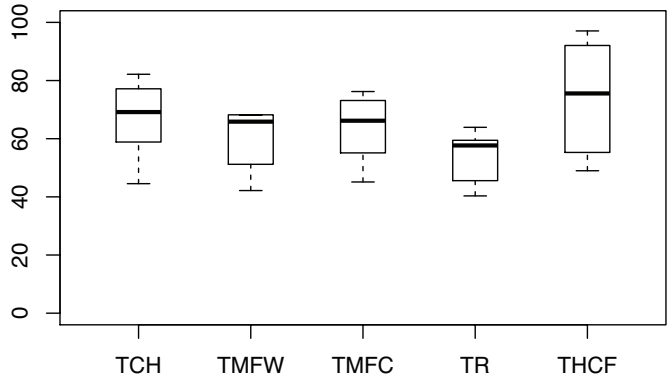

Fig. 7: nopCommerce - APFD Boxplots

Figure 8 shows the results for Coevery. When we compared the median values, our heuristic technique improved the rate of fault detection by 21% over the first three techniques on average and 41% over the random technique. Among the first three control techniques, T_{ch} and T_{tmfw} produced slightly better results than T_{mcf}.

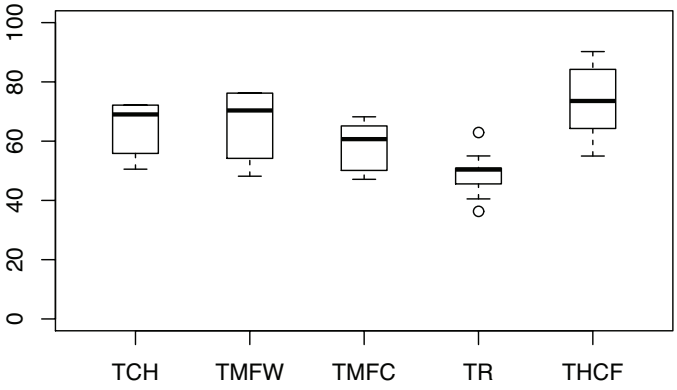

Fig. 8: Coevery - APFD Boxplots

Overall, the experimental results showed that our proposed recommender system-based technique performed better than all the control techniques across all three applications. Among the control techniques, the random performed performed worst across all applications, whereas the change history-based technique produced slightly better and more stable results for all applications.

B. RQ2 Analysis

RQ2 investigates whether the use of the recommender system can improve the effectiveness of test prioritization when we have a limited budget for testing. In RQ2 analysis, we measured NAPFD, which is a normalized ratio of APFD, when our resources were not consistent. In this experiment, first we executed 10% of our test cases, and we continued to execute the test cases in increments of 10% of the total until they had all been executed to see whether we could improve the fault detection rate given a time constraint dictating that running 100% of the test cases at one time was not feasible.

Table V shows the results of our three applications. This table shows the results of two primary analyses. The first part of the table presents the NAPFD scores on average, and the second part shows the improvement rates of the heuristic technique over the four other control techniques.

By examining the numbers in the table, we can observe that the improvement rates of our heuristic technique over the control techniques vary widely. When we compared the heuristic with T_{ch}, the improvement rates ranged from 4% to 41% for DASCP, from 17% to 63% for nopCommerce, and from 17% to 54% for Coevery. When compared with T_{mfw}, the improvement rates ranged from 11% to 132% for DASCP, from 42% to 140% for nopCommerce, and from 14% to 77% for Coevery, indicating significant improvements. When compared with T_{mfm}, the improvement rates ranged from 15% to 78% for DASCP, from 27% to 78% for nopCommerce, and from 27% to 48% for Coevery, indicating results similar to those for as T_{ch}. As for the comparison with T_r, the results

TABLE V: NAPFD Scores on Average.

Application	Test Exe.	Techniques					Improvement Rate over Control			
	Rate (%)	T_{ch}	T_{mfw}	T_{mfm}	T_r	T_{hcf}	T_{hcf}/T_{ch}	T_{hcf}/T_{mfw}	T_{hcf}/T_{mfm}	T_{hcf}/T_r
DASCP	10	18.54	12.17	15.12	14.33	23.37	26%	92%	54%	63%
	20	21.31	12.88	16.78	15.51	29.98	40%	132%	78%	93%
	30	28.86	21.19	25.12	18.47	40.95	41%	93%	63%	121%
	40	40.42	29.86	38.68	25.59	55.3	36%	85%	42%	116%
	50	54.34	36.21	42.9	39.15	67.07	23%	85%	56%	71%
	60	61.7	47.31	50.34	45.29	70.42	14%	48%	39%	55%
	70	68.34	56.96	65.97	60.74	76.67	12%	34%	16%	26%
	80	75.41	69.04	71.14	64.88	84.01	11%	21%	18%	29%
	90	83.35	77.28	77.69	67.11	89.83	7%	16%	15%	33%
	100	90.16	84.21	79.22	70.91	94.14	4%	11%	18%	32%
nopCommerce	10	17.54	12.17	15.75	8.33	28.14	60%	131%	78%	237%
	20	29.28	19.88	28.35	12.51	47.9	63%	140%	68%	282%
	30	38.86	27.19	35.45	15.57	55.28	42%	103%	55%	255%
	40	44.42	34.86	48.68	27.59	65.06	46%	86%	33%	135%
	50	58.34	39.16	54.94	35.91	74.78	28%	90%	36%	108%
	60	62.7	51.42	57.08	41.45	81.49	29%	58%	42%	96%
	70	68.14	55.03	59.97	50.02	86.38	26%	56%	44%	72%
	80	76.22	60.21	64.14	58.15	89.87	17%	49%	40%	54%
	90	79.4	66.01	70.22	60.17	95.14	19%	44%	35%	58%
	100	82.16	68.32	76.04	63.91	97.06	18%	42%	27%	51%
Coevery	10	26.54	23.17	29.12	20.33	41.02	54%	77%	40%	101%
	20	44.28	42.88	45.12	24.51	66.98	51%	56%	48%	173%
	30	60.86	44.19	50.12	28.57	73.28	20%	65%	46%	156%
	40	63.42	51.51	53.68	31.59	75.06	18%	45%	39%	137%
	50	64.34	55.74	58.3	39.62	77.1	19%	38%	32%	94%
	60	66.7	56.33	60.41	44.09	78.65	17%	39%	30%	78%
	70	68.19	59.86	62.97	46.11	81.13	18%	35%	28%	75%
	80	70.67	66.1	65.14	57.91	83.2	17%	25%	27%	43%
	90	71.81	73.14	66.03	59.01	86.69	20%	18%	31%	46%
	100	72.16	76.21	68.22	62.91	87.23	20%	14%	27%	38%

were more remarkable. The rates ranged from 32% to 282% for all three applications.

One outstanding trend we observed in the table is that the improvement rates are much higher when the time budget is smaller. For example, in the comparison with T_{ch} for nopCommerce, when 10% of budget was assigned, the improvement rate was 60%, but when we had a full budget, the rate dropped to 18%. A similar trend can be observed across all control techniques and applications. This indicates that our approach can be more helpful when companies are operating under a tight budget.

To visualize our results, we illustrate them in lineplots as shown in Figures 9, 10, and 11. Examining the lineplots for DASCP (Figure 9), we observe that the growth in NAPFD values for T_{chf} and T_{ch} trended upward quickly while running the first 50% of test cases, becoming nearly stable for the rest of the test cases. This means that more defects were detected at a relatively early stage of test execution when we applied these two techniques.

In the case of nopCommerce (Figure 10), we can see that in the first 20% test executions, NAPFD value produced by the heuristic approach (T_{chf}) increased dramatically. Also, there is a remarkable difference between the NAPFD scores of the heuristic technique and all the other control techniques at every stage of test execution. The most significant difference between scores occurred during the time that we were running between 20% and 30% of the test cases.

As for of Coevery (Figure 11), similar to nopCommerce, it is evident that after 10% of test case execution, the NAPFD value produced by the heuristic technique increased dramatically, but and after 20% of test case execution the increase was not as significant, in fact becoming stable.

We can infer that a great number of defects can be detected at a very early stage of test case execution when applying our approach to nopCommerce and Coevery, looking at the figures for those two applications.

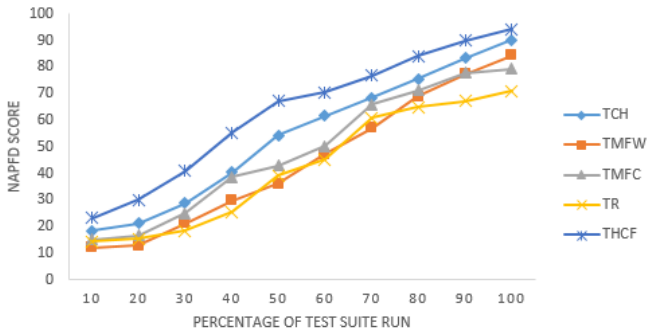

Fig. 9: DASCP- NAPFD lineplots

V. Discussion

In this section, we discuss the implications of results for researchers and practitioners as well as the limitations of recommender systems that we found during our experiment.

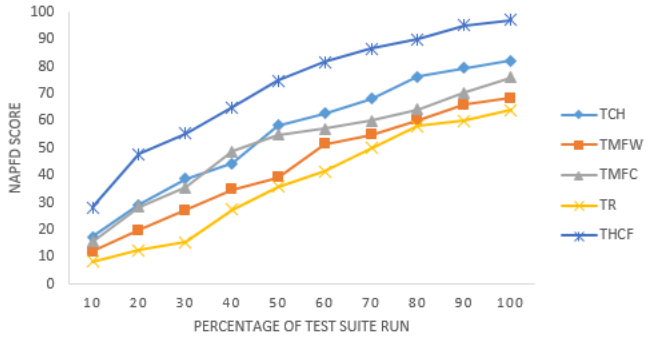

Fig. 10: nopCommerce- NAPFD lineplots

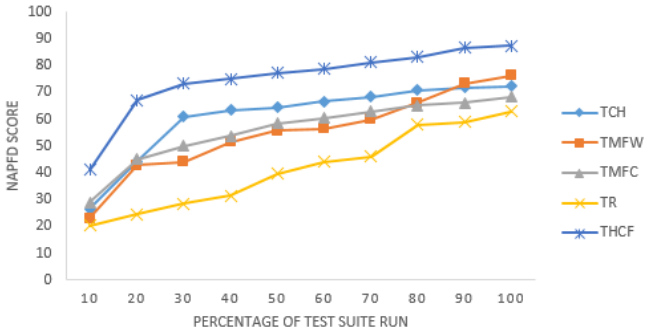

Fig. 11: Coevery- NAPFD lineplots

Implications of the Results: Our experiment results indicate that by utilizing a recommender system when we prioritize test cases, we were able to improve the effectiveness of test case prioritization. Further, we found that our proposed approach is far more effective than the control techniques when testers have a limited time budget during testing. In particular, from the RQ2 results, we learned that our approach produced much greater benefits when the time budget is limited. The findings from this study could help practitioners select appropriate test prioritization techniques considering their product delivery schedule and other circumstances (e.g., the availability of required software attributes).

We also found that the results showed some differences between two types of applications: proprietary and open source applications. When we compared DASCP, which is a proprietary application, with two open source application, we noticed that there is less data variance in DASCP. We examined our dataset to investigate the reason for this difference and we found that there is a higher cohesion between user interaction data in DASCP than in the two open source applications. We speculate that this difference came from the types of users who use the applications. DASCP users were actual users who have domain knowledge of the application, but for the open source applications, the users were student participants whose usage patterns were not coherent. This result indicates that the use of actual user data could produce more stable results than using volunteer testers, and this finding should be further investigated through additional experiments with actual user datasets.

Limitations of Applying Recommender Systems: While the results of our study indicate that recommender systems can help improve test case prioritization, like any other approaches, recommender systems also have their own limitations. In this study, we applied an item-based collaborative filtering algorithm that calculates the components' similarity based on user ratings. This means that our results are highly dependent on the accuracy of user ratings, and thus user ratings can greatly affect the experiment's results. There are three common limitations in collaborative filtering recommender systems; new user problem, new item problem, and sparsity [17]. Among these three limitations, two of them are related to our study.

- **Sparsity.** As we discussed earlier, we do not have a rating module in our applications, and thus we used access frequencies to each component as a user rating. Further, for the open source applications, nopCommerce and Coevery, we collected the user interaction data by asking non-professional users, so our data could contain noise and redundancy, which can affect the performance of our technique. Moreover, the number of collected ratings is relatively small, compared to the number of expected ratings to have an accurate prediction.
  Also, the distribution patterns of user ratings can affect the outcome of collaborative filtering algorithms. For example, in our case, some components were used by all users, such as registration and membership components, while some other components were ignored by the majority of users. In order to eliminate this issue, we need to collect more user interaction data by considering a larger number of users and a longer period of time during which to monitor user interactions. Further, having actual users would provide more realistic results, because their interactions would be based on real business functions and system workflow.
- **New Item Problem.** Another limitation of collaborative filtering that is related to our study is the "New Item Problem". It is a common practice that newly developed components are frequently added to a system. However, rankings on collaborative filtering algorithms are based on user access frequencies to the components. Therefore, the newly added components would not be in the recommender suggestion list until a certain number of users perform some tasks on them. In our study, in addition to using access frequency scores, we also applied change risk scores to recommend the highest risky components. Therefore, even if a new component has a high change risk score, its frequency score would still be zero, which makes the overall risk score zero. To overcome this limitation, we need to use a hybrid and normalized ranking score by assigning a small value to the components.

VI. 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.

VII. 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 [10], [34] 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 [15], [21], [30]. Although this approach is simple and nave, many empirical studies have shown that this approach can be effective [13], [14], [16], [27]. Recent prioritization techniques have used other types of code information, such as slices [19], change history [31], code modification information, and fault proneness of code [23].

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 [4]. Memon and Amalfitano performed a study in which they applied telemetry data to generate usage pattern profiles [8]. In another study, Amalfitano et al. built finite state models based on usage data that they collected from rich Internet applications [3]. Carlson et al. [9] presented clustering-based techniques that utilize real fault history information including code coverage. Anderson et al. [5] 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 [18].

Recommender Systems: 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 [17]. 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 [28]. For instance, Murakami et al. [25] proposed a technique that uses user editing activities detecting code relevant to existing methods. Christidis et al. [11] implemented a recommender system to display developer activities by using information artifacts with quantitative metrics. Danylenko and Lowe provided a context-aware recommender system to automate a decision-making process for determining the efficiency of non-functional requirements [12].

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.

VIII. 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.

References

[1] hhttp://www.nopcommerce.com/. [Accessed: Jan. 26, 2017].

[2] http://www.coevery.com/. [Accessed: Jan. 26, 2017].

[3] D. Amalfitano, A. R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *In Software Testing, Verification, and Validation Workshops*, pages 274–283. IEEE-ACM, 2010.

[4] J. Anderson, H. Do, and S. Salem. Customized regression testing using telemetry usage patterns. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016.

[5] J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. In *Working Conference on Mining Software Repositories (MSR)*, 2014.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. IEEE-ACM, 2006.

[7] Shawn A. Bohner. Extending software change impact analysis into cots components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, page 175. IEEE-ACM, 2002.

[8] P. A. Brooks and A. M. Memon. Automated gui testing guided by usage profiles. In *In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342. IEEE-ACM, 2007.

[9] R. Carlson, H. Do, , and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM '11 Proceedings of the 2011 27th IEEE International Conference on Software Maintenanc*, pages 382–391. IEEE-ACM, 2011.

[10] C. Catal and D. Mishra. Test case prioritization: A systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.

[11] K. Christidis, F. Paraskevopoulos, D. Panagiotou, and G. Mentzas. Combining activity metrics and contribution topics for software recommendations. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 43–46. IEEE-ACM, 2012.

[12] A. Danylenko and W. Lowe. Context-aware recommender systems for non-functional requirements. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, pages 80–84. IEEE-ACM, 2012.

[13] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. IEEE-ACM, 2008.

[14] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. 36(5), 2010.

[15] 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.

[16] S. Elhaum, A. G. Malishevsky, , and G. Rothermel. Test case prioritization: A family of empirical studies. 28(2):159–182, 2002.

[17] G. and A. Tuzhilin. Toward the next generation of recommender systems a survey of the state of the art and possible extensions. *IEEE Transactions on knowledge and Data Engineering*, 17(6):734–749, 2005.

[18] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *Proceedings of the 34th International Conference on Software Engineering*, pages 430–440. IEEE-ACM, 2012.

[19] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Int'l Comp. Soft. Appl. Conf.*, pages 411–420, September 2006.

[20] Y. Jiang, B. Cuki, T Menzies, and N Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18. ACM, 2008.

[21] 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.

[22] T. Lee, J. Nam, D. Han, S. Kim, and H. Peter. Micro interaction metrics for defect prediction. In *ESEC/FSE '11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321. IEEE-ACM, 2011.

[23] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases on Baysian Networks. In *Found. App. Softw. Eng.*, pages 276–290, March 2007.

[24] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE '08 Proceedings of the 30th international conference on Software engineering*, pages 181–190. IEEE-ACM, 2008.

[25] N. Murakami, H. Masuhara, and T. Aotani. Code recommendation based on a degree-of-interest model. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*, pages 28–29. IEEE-ACM, 2014.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05 Proceedings of the 27th international conference on Software engineering*, pages 284–292. IEEE-ACM, 2005.

[27] X. Qu, M.B. Cohen, and G.Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 75–85. IEEE-ACM, 2008.

[28] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering*. Springer, 2014.

[29] G. Rothermel, R. Untch, C. Chu, , and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179–188. IEEE-ACM, 1999.

[30] 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.

[31] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 81–90, November 2007.

[32] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *FSE '12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[33] R. Robillard Walker and T. Zimmermann. Recommendation systems for software engineering. In *Proceedings of the 27th International Conference on Software Engineering*, pages 80–86. IEEE Press, 2010.

[34] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[35] M. Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. In *In Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.