2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 978-1-6654-4456-9/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSTW52544.2021.00035

A Tag-based Recommender System for Regression Test Case Prioritization

Maral Azizi
Departemant of Computer Sceince
East Carolina University
azizim19@ecu.edu

***Abstract*—In continuous integration development environments (CI), the software undergoes frequent changes due to bug fixes or new feature requests. Some of these changes may accidentally cause regression issues to the newly released software version. To ensure the correctness of the newly released software, it is important to perform enough testing prior to code submission to avoid breaking builds. Regression testing is one of the important maintenance activities that can control the quality and reliability of modified software, but it can also be very expensive. Test case prioritization can reduce the costs of regression testing by reordering test cases to meet testing objectives better. To date, various test prioritization techniques have been developed, however, the majority of the proposed approaches utilize static or dynamic analyses to decide which test cases should be selected. These analyses often have significant cost overhead and are time consuming. This paper introduces a new method for automatic test case prioritization in a CI environment intending to minimize the testing cost. Our proposed approach uses information retrieval to automatically select test cases based on their textual similarity to the portion of the code that has been changed. Our technique not only helps developers to organize and manage the software repository but also helps them to find the relevant resources quickly. To evaluate our approach, we performed an empirical study using 37 versions of 6 open source applications. The results of our empirical study indicate that our proposed method can improve the effectiveness and efficiency of test case prioritization technique.**

Keywords: Regression Testing, Test Case Prioritization, IR-based Regression Testing, Recommender Systems, Continuous Integration, Tag-based Recommender System.

I. Introduction

Test case prioritization is a technique that improves the effectiveness of regression testing by exercising more important test cases earlier [51]. Most common prioritization techniques often require collecting multiple sources of information from source code such as code coverage, fault history, code structure, etc. Although, many empirical studies have shown the effectiveness of these techniques [18], [8], [45], [28], [66], [62], [61], [36], [47], [21], [68], they suffer from several limitations. One major limitation of these techniques is that this information is usually unavailable before test cases are executed. Moreover, collecting this information requires a significant amount of time and has to be repeated for each new version of a software; therefore, previously collected information is not accurate for a new version of the program due to the changes in source code and test suite.

In practice, to perform regression testing, practitioners usually select test cases for execution based on their prior knowledge or experiences. Although, those selections are useful because they are made by a human expert, there would be at the risk of ignoring regression defects because often practitioners do not have the information of the corresponding code changes made to the product [52]. These limitations can be more problematic in a continuous development environment because the speed of software delivery has become one of the important success factors for these applications. To meet this demand, efficient testing approaches that can be aligned with today's incremental and iterative software delivery practices should be considered.

Over the past years, various recommendation frameworks have been developed to solve the software engineering problems [20], [49], [39], [65], [33], [46]. While these frameworks cover a wide range of application domains, to the best of our knowledge, a recommendation framework for regression testing was still lacking. Therefore, in 2018, we have started developing a collaborative filtering recommender system for regression testing [13]. In the initial development phase of the framework, we mainly focused on analyzing users' interactions with the systems. Later, we proposed a graph-based network in which the nodes of the network represents test cases and the edges represent the textual similarity between the test cases [14]. In that study, the recommendation system forms a query based on the code changes from two consecutive versions of a program and recommend test cases based on the textual similarity between the code changes and test cases.

While our previous methods succeeded to recommend test cases that can detect regressions, their performance was limited to the accuracy of term similarity calculation. This means that if a term in the modified portion of the code was not matched with the terms in the body of a test case, that test would not be selected for execution. We showed that there were cases in which a test method did not include the exact key terms of the query but it was a fault revealing test case. Since the performance of the previous techniques was heavily rely on the term matching, it can be a serious threat to validity.

To address the above mentioned limitation, in this paper, we propose a tag-based recommender system for regression test case prioritization (RTP). Our approach utilizes topic modeling to generate tags for given test cases. It then stores and indexes test cases as a set of tags in a database. Our approach helps

978-1-6654-4456-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSTW52544.2021.00035

practitioners in two major ways: first, system tagging helps developers to organize the source files, second, it enables practitioners to find relevant resources quickly, which reduces the cost of regression testing significantly. To recommend test cases, our tag-based recommender system uses the modified portion of the code to form a query, and test cases as a document collection. It then recommends the most relevant test cases to a given query based on the tags affinity between the documents and the queries. This paper is the first study of tag-based recommendation system in regression testing that describes all processing steps, discusses the retrieval performance, identifies the limitations, and suggests alternative solutions for the limitations of information retrieval-based (IR-based) RTP. We also discuss how our approach can improve not only the retrieval accuracy but also RTP performance. We describe the main computational steps involved, from data acquisition and preprocessing, to tag generation and feature selection, and finally to the ranking.

To evaluate the effectiveness of the proposed approach, we used 37 versions of 6 open source software projects written in Java and C# and compared the performance of our method against five widely used test case prioritization techniques. The results of our empirical study show that our approach can be effective in practice by reducing a significant amount of time compared to the common regression testing techniques such as code coverage-based techniques. Thus, our tag-based recommender system provides an effective alternative approach to addressing the prioritization problem without requiring any dynamic coverage or static analysis. The main contributions of this research are as follows:

- We introduce a new cost effective regression test case prioritization technique. Our proposed technique is language independent, fast, scalable, and light-weight, which eliminates the cost of coverage profiling.
- Our study also empirically evaluates the proposed approach by comparing it with five test prioritization techniques, which are commonly used for regression testing.
- We discuss how tag-based recommender system can improve the recall and precision by describing all the steps, which includes: data collection, preprocessing, document construction, feature selection, and tag generation.

The rest of the paper is structured as follows. Section II explains the proposed approach. Section III outlines the research questions and the details of the experimental design. Section IV presents the results of our study, and Section V discusses the results including practical implications and guidelines for testers. Section VI discusses the threats to validity. Section VII presents related work, and Section VIII discusses conclusions and future work.

II. Approach

The goal of our approach is to improve the effectiveness as well as the efficiency of test case prioritization. To achieve this goal, we propose an automated prioritization technique that utilizes two sources of information (program change information, and program source code). Using this information, we build a tag-based recommender system that automatically recommends test cases based on their relevance to the portion of the program that has been changed. Figure 1 presents an overview of the proposed technique, which includes four major steps: tag generation, query construction, system tagging, and recommendations. First, the proposed approach uses tests source files to construct documents. To build the document collection, we prepossess the files, index them and store the indexed elements into a database (the details are explained in Section II-A1). Using the code change information between two consecutive program versions, we build queries to identify a set of test cases that likely exercise the modified portion of the program, which can increase the chances of regression fault detection. We then generate tags from the constructed documents. To generate the tags, our recommender system creates a set of topics using Latent Dirichlet Allocation (LDA) technique and suggests tags for test cases based on their relevance to each topic(the details are explained in Sections II-A2). In the following subsections, we describe our proposed approach in detail including the tag generation, and test recommendation with a walk-through example.

A. Tag Generation

In IR-base RTP, the retrieval algorithm will return the test cases based on their textual similarity to the queries [53], [14]. However, test methods and queries are often short and contain few non-discriminatory terms, which can lead to low retrieval accuracy. Tagging systems allow users to annotate and classify content, therefore, finding relevant content would be easier for the users. In the following subsections, we explain the steps that are required to generate tags.

1) Document Construction: This step transforms the raw data into a format that will be more effective for the retrieval process. Document construction often consists of feature extraction, followed by the construction of an appropriate data structure that is suitable for retrieval and indexing. Constructing documents usually depends on the information granularity and the choice of IR technique. Unit tests are often a collection of source files, where each file consist of one or more test classes and test methods. In this research, we construct the documents in test method level of granularity because the previous studies show that it is more effective than test class level in term of fault detection [53], [25], [23], [24].

To build the documents, in this work, we first collect all test files from each program version. We then tokenize each term in each document and remove stop words, mathematical operators, and all symbols. In this study, stop words are generic programming terms (i.e. $public, private, if, else$, etc.). We also do not remove comments or identifier names because studies showed that they are particularly important from the information retrieval point of view [53], [14].

2) Tag Generation Using LDA: Most commonly used datasets for tag-recommendation systems contain a set of initial tags that can be created manually by the users. These systems after receiving the initial tags, apply different tech-

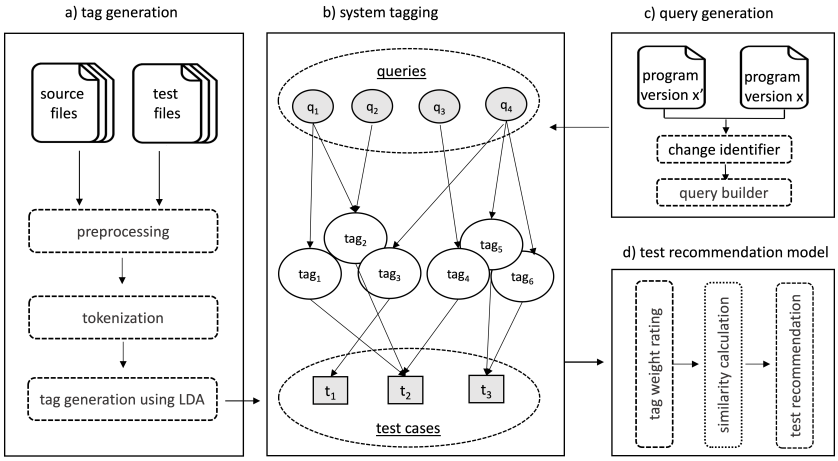


Fig. 1: An Overview of Tag Recommendation Framework

niques such as semantic analysis, pseudo feedback, etc. to incorporate more relevant tags to a given content [26], [29], [64], [38]. One way to generate tags in this context would require developers to look into the files and manually assign the appropriate tags to each document, however, this will add an extra burden on the software developer. Because one of our goals is to reduce the cost of regression testing we apply *Tf-Idf* formula to automatically identify the most repetitive terms in the entire set of documents as a set of initial tags. This technique have been investigated in many studies and the results show it is effective in information extraction by assigning a weight to keywords in a document [9]. However, since the tags obtained from *Tf-Idf* are restricted to a certain vocabulary in a given document, developers can select any tags they like to annotate the test files. Thus these tags can be inconsistent and may not represent the content of test cases. This reduces the usefulness of tags, in particular for documents annotated by only a few practitioners.

To increase the retrieval accuracy, we use LDA to elicit latent topics from the resources. In this context where the system annotates the resources automatically, the resulting topics reflect shared usability of the documents and the tags of the topics reflect a common set of vocabulary to describe the document. More generally, LDA helps to explain the similarity of data by grouping features of this data into unobserved sets. A mixture of these sets then constitutes the observable data. The LDA method initially was introduced by Blei et al. [16] and in this study, we apply this technique to generate relevant tags for our documents. LDA has been investigated with many researchers to solve a variety of software engineering problems, e.g. analyzing source code evolution [56], [54], program comprehension [15], traceability link recovery [12], etc. Moreover, LDA is fast and scalable, which maintains the RTP overhead to a minimum.

LDA finds a mixture of topics for a given document d with each topic z described by terms following another probability distribution. This process can be formalized as

$$P(t_i|d) = \sum_{j=1}^{Z} P(t_i|z_i = j)P(z_i = j|d)$$

where $P(t_i|d)$ is the probability of the ith term for a given document d and z_i is the latent topic. $P(t_i|z_i = j)$ is the probability of t_i within topic j. $P(z_i = j|d)$ is the probability of selecting a term from topic j in the document. To use LDA, we need to assign appropriate values to its parameters: the number of topics, K; the document-topic smoothing parameter, α; the topic-word smoothing parameter, β; and the number of Gibbs sampling iterations to execute. LDA assigns to each document latent topics together with a probability value that each topic contributes to the overall document. In this study, the test files are resources $r \in R$, and each resource is described by initial tags $t \in T$ that is assigned by extracting most frequent terms using *tf-idf* formula. Therefore, for each resource r we have some bookmarks $b(r, t_i)$ assigned by the system. We then can represent each test document in the system not only with its actual tags but also with the tags from topics discovered by LDA. Common values for K in software engineering studies varies from 5 to 500 topics, depending on the number of documents, granularity desired, and task to perform [55]. In this study, we set $K = 50$ (We investigate parameter sensitivity in Section V-A). We use the LDA package default values for α, β, and the number of iterations, which are 0.1, 0.1, and 200, respectively. Note that these values are not dependent on the number of documents in the SUT.

B. Query Generation

Constructing queries is a critical task in information retrieval systems. Many studies show that different types of query construction may result in different retrieval outcomes [32], [31], [46], [33], [30]. The content and the quality of the queries may depend on multiple factors such as retrieval goal, retrieval

performance, results accuracy, etc. Because the goal of our study is to identify the most effective test cases in terms of regression fault detection capability, we define queries as the program differences between two versions at the line level.

To construct the queries, we collect the code changes between two versions of a program by applying a *diff* tool [1]. Before we apply the diff tool, we remove spaces, blank lines, symbols including mathematical signs, parentheses, brackets, etc. from each file. We also perform term filtering; (i.e., deleting stop-words). Note that stop words in this context are different from standard English stop words. In this research, we remove the most frequent reserved terms in Java and C# languages (i.e. $public, private, if, else$, etc.). After preprocessing is complete, we build the queries, which is a vector of tokens.

C. System Tagging

Part (b) of Figure 1 shows the system tagging process. In this step, we present each test case d_i as vectors of tags instead of documents composed of terms. Each test is represented as a vector of tags with a probability value. For each document d we initially assign some tags extracted by *Tf-Idf* formula. Then we add new tags from the topic discovered by LDA to enrich the documents.

For the smaller test files where we only have a small number of tags ($i \in 1...10$), we can expand the representation of these resources with the top tags of each latent topic. Probabilities are assigned not only to the latent topics for a single resource but also to each tag within a latent topic to indicate the probability of this tag being part of that particular topic. We represent each test d_i as the probabilities $P(z|d_i)$ for each latent topic $z_j \in Z$. Every topic z_i is represented as the probabilities $P(t|z_j)$ for each tag $t_n \in T$. By combining these two probabilities for each tag for d_i, we get a probability value for each tag that can be interpreted similarly as the relative tag frequency of a test case. We set the threshold for the number of tags associated with each document to 20 (we investigate parameter sensitivity in Section V-A). We then expand the initial queries by repeating the same process. After creating the initial queries, the top 10 features from the LDA will be selected for the query expansion. The underlying hypothesis of this technique is that since the selected features have the highest retrieval scores, they are likely to contain further informative terms that are related to the original query [19].

D. Test Recommendation

1) Tag Weight Rating: A common practice in term reweighting is to give more importance to the original terms. Subsequently, the document d is reformulated using the following interpolation formula:

$$w'_{t,d'} = (1 - \lambda)\,.\,w_{t,d} + \lambda\,.\,Score_t$$

where d' is the expanded document, d is the original document, λ is a parameter to weight the relative contribution of document terms and expansion tags, and $Score_t$ is a weight assigned to expansion term t. λ is an experimental parameter that determines to what extent one should mix the original document with new tags by their associated probabilities. The value of λ can be adjusted based on the data. In this study, we assigned λ = 0.8. Section V-A describes the effects of different λ values on the retrieval accuracy.

2) Similarity Calculation: Assume that documents and a query are shown as below:

$$D = \{d_1, d_2, d_3, ..., d_n\}$$
$$Q = \{q_1, q_2, q_3, ..., q_n\}$$

where each element i in D and Q represent the term frequency of $term_i$ in document D and query Q, respectively. Further, to obtain the final results a query is used to retrieve the data from a set of documents using the below formula:

$$Sim(D, Q) = \sum_{i=1}^{n} tf_D(d_i)\; tf_Q(q_i)\; idf(t_i)^2$$

where $tf_D(d_i)$ is the term frequency of $term_i$ in the document d, $tf_Q(q_i)$ is the term frequency of $term_i$ in the query q, and $idf(t_i)$ is the inverse document frequency of $term_i$ in the collection.

E. An Example Tag Recommendation and Query Retrieval

Figure 2 shows an example of the tag recommendation process. The left side of the figure represents a source file and the right side of the figure represents three test cases, and a code commit of *Joda-Time* program [1]. In this example, all three test cases exercise the "$DateTimeComprator$" class that is represented on the left side of the Figure 2.

The bottom section of this figure shows the query vector that is constructed by following the instruction explained in Section II-B; a set of initial tags that are extracted using *Tf-Idf* formula and recommended tags are assigned using LDA technique. The query contains four initial key words ("ilowerlimit", "datetimecomparator", "iupperlimit", "getname") that we stemmed each word to its root form (e.g. getName to getname). The three common words if, $else$, $return$, and symbols are also filtered. The other 10 tags in the query vector that are highlighted in red are added from LDA topics to enrich the query since it is very short. The button section of Figure 2 shows test cases A, B, and C that are represented as a vector of tags along with their probability.

The initial tags are the tags that are extracted from the body of the test cases using *Tf-Idf* formula and the recommended tags are extracted from LDA topics. In a standard text retrieval scenario, in this example, only test case A contains a key-term from the query ("$DateTimeComprator$") and the other two test cases B and C do not contain any key-term from the original query vector. Therefore, using standard text retrieval where the documents are extracted based on the exact term matching, the two test cases B and C would be dismissed in the retrieval process despite the fact that they exercise

[1]source code and test cases of this program can be found at https://github.com/JodaOrg/joda-time

the same portion of the code. However, using the proposed technique, where the test cases are stored by a set of associated tags in a database, all three test cases will be selected in the recommendation phase.

III. Empirical Study

In this paper, we investigate whether the use of proposed approach can improve the effectiveness of test case prioritization techniques. To assess our proposed technique, we performed an empirical study. The following subsections present our objects of analysis, data collection, study setup, and threats to validity. In particular, we address the following research questions:

RQ1: Is our proposed technique effective in identifying relevant test cases to the changed files?

RQ2: How does our proposed approach perform compared to the existing techniques in terms of fault detection?

A. Objects of Analysis

In this study, we used six open source programs. Table **??** lists the applications and their associated data: "Version pair" (the program versions that we used for the experiment), "LOC" (the number of lines of code), "TstMethod" (the number of test cases in method level), "TstClass" (the number of test cases in class level), "Faults" (the number of faults), "Queries" (the number of queries that we built from the program change files), "SrcToken" (the number of tokens of source code), and "TstToken" (the number tokens of test files).

nopCommerce is an open source e-commerce shopping cart web application built on the ASP.Net platform [2]. *Umbraco* is a large-scale open source content management system, which has been written in $C\#$ language [3]. We use *XStream* as a subject program for our study because this program have been used in multiple IR-based RTP [52], [58], therefore, we can have a better comparison of the effectiveness of our proposed technique. Three other applications were obtained from the defects4j database [4]: *JFreeChart*, *Joda-Time* and *Commons-Lang*, which are written in Java. All the faults for *JFreeChart*, *Joda-Time* and *Commons-Lang* are real, reproducible, and have been isolated in different versions. The faults used in *nopCommerce* and *Umbraco* have been reported by users [2]. Also, all applications have multiple consecutive versions and test cases. In total, 37 versions of 6 programs were used to create the data required for evaluating the proposed technique.

B. Variables and Measures

1) Independent Variables: The independent variable in this study is regression test prioritization technique. We considered five test case prioritization techniques, which we classified them into two groups: control and heuristic. For our heuristic techniques, we use the approach explained in Section II, so, here, we only explain the control techniques.

[2]The reported faults for *nopCommerce* are available in the *GitHub* repository [5], and the bug history for *Umbraco* is accessible through their website [6].

1) Untreated (T_1): This technique executes test cases based on their original order.
2) Random (T_2): This technique prioritizes test cases randomly. We repeat this technique 30 times for each program.
3) Total Statement (T_3): This technique prioritizes test cases based on the total number of statements they cover. If multiple tests cover the same number of statements, they are ordered randomly.
4) Additional Statement (T_4): This technique prioritizes test cases based on the number of additional statements they cover. If multiple test cases cover the same number of statements not yet covered, they are ordered randomly.
5) Text Retrieval Based (T_5): This technique applies standard IR to prioritizes test cases. To measure the correlation between test cases and code change, in this technique we use the *Tf-Idf* formula. For further details about the implementation of this technique, please refer to Saha et al. [52].

2) Dependent Variable and Measures: Dependent variables for RQ1 are Precision, Recall, and F-Measure. For RQ2 the dependent variable is average percentage of fault detection (APFD), which measures the average percentage of faults detected during the test suite execution. The range of APFD is from 0 to 100; higher values indicate faster fault detection rates. 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 [50] as follows:

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

C. Data Collection and Experimental Setup

As described earlier, our proposed technique does not require any dynamic or static code coverage information and it calculates the similarity between code commits and source code/test cases. To collect code coverage information for control techniques we used Visual Studio Test Analysis plugin for *nopCommerce* and *Umbraco-CMS*. For other four Java applications, we used JaCoCo plugin on netbeans IDE [7]. We also collected test execution time using a PC with CPU Core i7, memory 16 GB, and operating system Windows 10.

IV. Data and Analysis

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

A. RQ1 Analysis

RQ1 considers whether the proposed approach is effective in identifying the test cases that are corresponding to the portion of the program that has been modified. As described in Section II-C we can set a threshold for the probabilities up to which we recommended tags. Depending on the resource availability and budget, the recommendation algorithm can control the number of tests to be selected for a given

Fig. 2: An Example of System Tagging and Query Extraction

No	Object	Version pair	LOC	# TstMethod	# TstClass	# Faults	# Queries	# SrcToken	# TstToken
P_1	nopCommerce	2.00 - 2.10	133,862	523	126	28	65	17,803	2,768
P_2	nopCommerce	2.20 - 2.30	141,232	585	136	32	118	18,052	2,893
P_3	Umbraco-CMS	7.7.0 - 7.7.2	312,511	6,025	628	41	97	26,700	14,552
P_4	Umbraco-CMS	7.7.4 - 7.7.5	312,886	6,026	628	28	131	26,811	14,546
P_5	Joda-Time	0.9 - 0.95	17,818	923	80	10	143	3,328	2,767
P_6	Joda-Time	0.98 - 0.99	20,317	3,181	218	10	128	4,905	3,765
P_7	Joda-Time	1.1 - 1.2	22,149	3,360	237	21	130	5,290	4,368
P_8	Joda-Time	1.2 - 1.3	24,001	4,256	254	21	115	5,380	4,622
P_9	Joda-Time	1.3 - 1.4	25,292	4,511	263	12	54	5,496	4,812
P_{10}	Joda-Time	1.4 - 1.5	25,795	4,701	266	18	112	5,610	5,106
P_{11}	JFreeChart	1.0.0 - 1.0.1	95,669	1,512	225	12	98	14,219	6,039
P_{12}	JFreeChart	1.0.2 - 1.0.3	101,713	2,151	231	10	6	11,855	4,750
P_{13}	JFreeChart	1.0.4 - 1.0.5	74,434	2,360	351	13	101	12,057	5,102
P_{14}	JFreeChart	1.0.6 - 1.0.7	76,065	2,680	366	8	85	12,662	5,522
P_{15}	JFreeChart	1.0.8 - 1.0.9	81,541	2,738	393	11	186	12,738	5,636
P_{16}	Commons-Lang	3.02 - 3.03	19,705	1,698	83	17	164	6,732	3,492
P_{17}	Commons-Lang	3.03 - 3.04	21,132	1,703	83	14	123	7,256	3,866
P_{18}	XStream	1.30 - 1.31	15,983	885	150	10	156	3,740	2,563
P_{19}	XStream	1.31 - 1.40	16,280	924	140	13	178	3,824	2,662
P_{20}	XStream	1.41 - 1.42	16,851	1,200	157	11	206	4,103	2,893

TABLE I: Subject Applications Properties

query. The outcome of the recommendation is affected by the probability value that defines the initial set of tests, which is used for selecting the final set of tests considering the testing budget. Theoretically, the probability value ranges between 0 and 1. By setting the probability value close to 0, we have to consider almost all test cases for the execution. Conversely,

by setting the probability value close to 1, almost no tests will be considered for the selection. Thus, finding a reasonable and practical probability value is an important step for our approach. To examine the effect of the probability value on identifying the corresponding test cases we set five different thresholds which are shown in Table II.

Table II shows the average precision, recall, and f-measure (FM) of the correctly recommended test cases for a given query. We obtain maximum f-measure when we set the probability threshold to 0.01. Similarly, Table III shows the accuracy results for our baseline technique. In this technique, test cases are recommended based on their textual similarity to a given query. To measure the textual similarity, in this technique we use *Cosine Similarity* formula [34]. We examine the results of recommendation by setting multiple similarity thresholds that are shown in the first column of the Table III. As can be seen from this table, precision decreases when lowering the threshold whereas recall increases. Similar to our proposed approach, we obtain maximum f-measure at 0.199 when we set the similarity threshold to 0.01.

Our quantitative results from Table II and III show that the proposed approach leads to significant improvements over the standard IR technique. Overall, the precision goes up 120% compare to the baseline technique for all cases. The improvements obtained by the proposed approach are noticeably higher than those obtained by the standard IR method particularly for the probability threshold of 0.10 ($\sim$ 121%, 146%, and 139%, increase on precision, recall, and FM respectively).

Prob Threshold	Precision	Recall	F-Measure
0.15	0.651	0.117	0.198
0.10	0.602	0.227	0.329
0.05	0.453	0.326	0.379
0.01	0.398	0.473	0.432
0.005	0.355	0.529	0.424

TABLE II: Retrieval accuracy results for query extraction using proposed technique with 20 tags with different probability thresholds to recommend a tag for test cases

Sim Threshold	Precision	Recall	F-Measure
0.15	0.317	0.074	0.119
0.10	0.273	0.092	0.137
0.05	0.225	0.171	0.194
0.01	0.162	0.259	0.199
0.005	0.140	0.313	0.193

TABLE III: Retrieval accuracy results for query extraction using standard IR technique with different similarity thresholds to recommend a test case for a given query

B. RQ2 Analysis

Our second research question investigates whether the proposed technique can improve the rate of fault detection for unit test cases applied to our subject programs. Figure 3 represents the average percentage of the fault detected for each program. The horizontal axis of each figure shows the prioritization techniques and the vertical axis shows the percentage of the mean APFD values for each program. This experiment was performed 30 times with random selections each time and the results shown are the average across all runs. APFD values are computed by prioritizing all test cases for each application.

As can be seen in Figure 3 the APFD values obtained by the proposed technique are higher in most cases. In five out of six subject programs, prioritization by the proposed technique yielded a higher APFD value than the code coverage based technique. The proposed technique outperformed the standard IR technique (T_5) in all cases. Among all examined control techniques, overall, the additional coverage technique (T_4) is slightly more effective than the others. Particularly, in cases of *JFreeChart*, additional code coverage based technique (T_4) performed better than other techniques. Looking into the characteristics of this program that are shown in Table **??**, in two versions of this program (P_{12}) the number of code changes is very small (6 queries), which is the cause of the poor performance of text retrieval based techniques (T_5 , T_6).

In particular, this difference was more outstanding for the applications with larger test suite size (*Umbraco* and *Joda-Time*). For instance, in the case of *Umbraco*, the APFD value increased by 16%, 13%, 35%, 95%, and 121% compared to T_5, T_4, T_3, T_2, and T_1, respectively. We speculate that the larger number of test cases and queries resulted in a richer corpus; consequently, improved the performance of the proposed technique. Based on this observation we can conclude that the application of the tag-based recommender system leads to substantial improvement in regression test case prioritization.

V. Discussion

To further explore the results of our experiment, in this section, we discuss our findings that can serve as the practical guidelines for test prioritization.

A. Parameter Sensitivity Analysis

The recommendation algorithm takes three input parameters: the number of tags to represent a test case, the overall number of latent topics to be identified in the given corpus (in this study corpus is the test suite of each program), and the interpolation parameter (λ). As explained in Section II-A2, we set the number of LDA topics to a constant values of 50, and we limited the associated tags of each test case to 20. We also set the interpolation parameter (λ) to a constant value of 0.8. In this section, we want to examine how different values of the number of tags, number of latent topics, and λ can impact the accuracy of the retrieval.

Object	# LDA Topics			
	25	50	100	200
nopCommerce	0.483	0.504	0.478	0.464
Umbraco-CMS	0.362	0.380	0.393	0.390
Joda-Time	0.312	0.302	0.285	0.272
JFreeChart	0.209	0.231	0.223	0.216
Commons-Lang	0.292	0.272	0.246	0.231
XStream	0.288	0.280	0.278	0.262

TABLE IV: F-measure for different applications and different number of LDA topics (threshold 0.01)

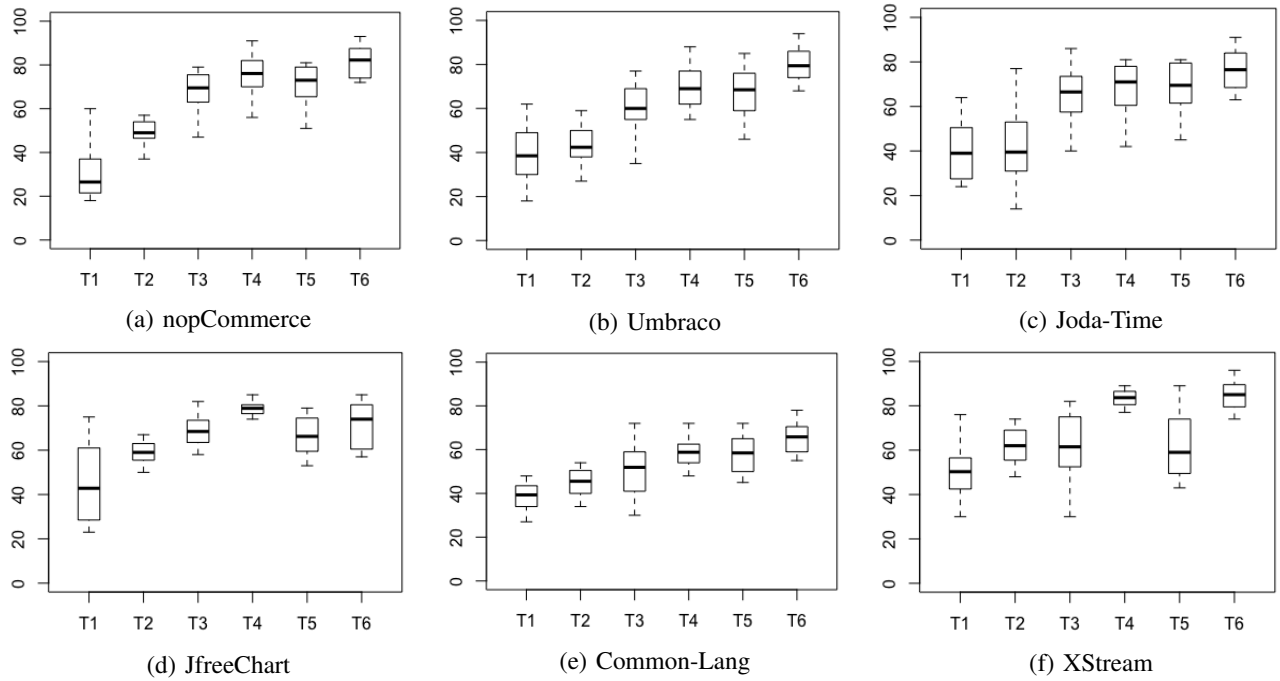
The results for varying the number of latent topics are shown in Table IV. The f-measure is shown for 25, 50, 100, and 200 latent topics. As can be seen in the table, F-Measure decrease with the increase in LDA topic size for *Joda-Time, Common-Lang, nopCommerce*, and *XSTream*. This effect is reversed in case of *Umbraco-CMS*. By looking at the characteristics of *Umbraco-CMS* we can see that the size of this application is significantly larger than other subject programs. Therefore, we can conjecture that higher number of latent topics leads to better performance for a larger corpus. Typically, a small number of topics leads to fairly general topics that are mixtures of more specific subtopics. Utilizing generic topics, in most cases increases the precision. However, it is more beneficial to separate the generic topics into more specific topics. Our results indicates that 50 LDA topics give the best average results.

In order to examine the effectiveness of the recommended tags, we compared the ranking performance for varying the number of assigned tags to each test case. To evaluate the effectiveness of the recommended tags we use standard measure, Mean Average Precision (MAP), which is used in many IR-based studies [19]. In this study, effectiveness refers to the best rank obtained by any of the resources relevant to a query. MAP is a measure of the accuracy of a retrieval approach based on the average precision of each query q in the set Q. Given R_q, the set of resources relevant to query q, the average precision is computed as the average of the precision values at the resulting rank of each document. MAP is the mean of the average precision of the set of queries Q, defined as follows:

$$MAP(Q) = \frac{1}{Q} \sum_{q \in Q} \frac{1}{R_q} \sum_{r \in R_q} Precision(rank(r))$$

If no relevant test case is retrieved, precision is taken to be 0. Figure 4 shows the MAP for each application. We examined the performance of the ranking by assigning different number of tags to each test case from the range of 5 to 50. The similarity threshold for tag recommendation was set to 0.1. From the figure, we can observe that the improvement rates vary widely. As can be seen from Figure 4, we obtain the best results for most applications when we assigned tags in the range of 10 to 20. Comparing the results for 10 assigned tags with other values, the improvement rates ranged from 12% to 73% for all subject programs. As for the comparison with 20 assigned tags, the results were more remarkable with 85%, 19%, 41%, and 54% improvement compared to 5, 30, 40, and 50 tags for all SUTs.

Further to investigate the impact of the parameter setting we run the experiment by assigning different values to λ. The parameter λ determines the weight to be accorded to the new terms to be added to a query in relation to the original terms. Figure 5 shows the impact of this parameter on the retrieval performance. As can be seen from Figure 5, the MAP reaches its peak point for λ = 0.8 for *nopCommerce* and *XStream*. One outstanding trend we observed in the table is that in all subject programs, the proposed technique achieved better accuracy for



(a) nopCommerce (b) Umbraco (c) Joda-Time

(d) JfreeChart (e) Common-Lang (f) XStream

Fig. 3: APFD box plots, all programs, all techniques. The horizontal axes list techniques, and the vertical axes denote APFD scores.

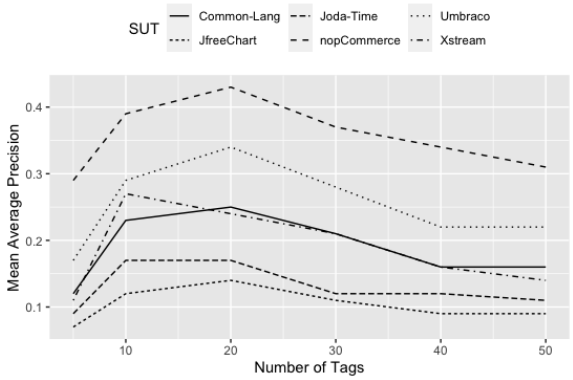


Fig. 4: Mean Average Precision for test recommendation with different number of tags

higher values of λ. From this observation, we can conclude that the initial tags are slightly more effective in representing test cases than extended tags.

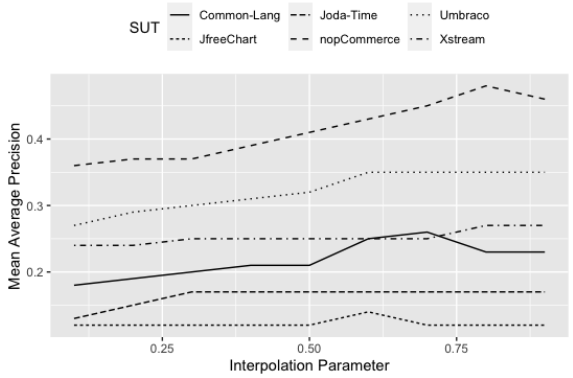


Fig. 5: Mean Average Precision for test recommendation with different interpolation values

B. Limitations

Although the proposed approach and our empirical results are promising, our approach has some limitations that we want to address. In our study, we applied a textual similarity measurement to estimate how much each test can cover the changes in a file. This means that our results are highly dependent on the quality of program naming and similar terms in source code and corresponding test cases. For instance, if a variable/method name does not imply its actual usage, the recommended tags might not reflect the functionality of the method, which might lead to extracting irrelevant resources. However, our quantitative results show that the probability of such occasions is very low.

Another limitation of the proposed approach is that the test recommendation process acts like a black box that may complicate the interpretation of the logic used by the system to deliver results. For instance, some of the recommended test cases might not contain the terms in the query. For example, a test may be returned because the anchor texts pointing to it contain the query terms, or because a query term is derived by a more general term. In this scenario, a user might find the results inadequate. One way to reduce the lack of transparency and increase user control over the relationships between query and results is to display the search results and allow some form of manipulation of query and results by the user. For example, revising the assigned tags, adjusting the weight of each tag based on the practitioners' point of view, etc. Customization of the recommendation results could be investigated in the future.

C. Cost-Benefit Analysis

To perform the cost-benefit analysis, we divided the costs for applying the proposed technique in two categories: 1) initial cost, and 2) incremental cost. The initial cost in this study is the cost of document construction (e.g., creating tags and setting properties values), and the cost of tool implementation. Once we collect the required data and build the tool, we do not need to repeat this process again. The incremental cost in this study includes updating the database (adding new tests, tags, and their properties) and building new queries. This cost varies on the frequency of code changes. The incremental cost for most applications is negligible. For instance, updating the database from version 0.95 to 0.99 of *Joda-Time* took less than 6 seconds, which has the largest growth of the number of test-methods in our study. For this specific case, it took 51 seconds to extract test methods for 128 queries. On the other hand, for the same project (P_6), the additional coverage technique took 40.23 minutes. This cost includes 37.06 minutes for profiling and 197 seconds for test prioritization. The time complexity of the proposed approach grows linearly as the size of the test suite grows, while the additional strategy grows quadratically [45]. Therefore, we can conclude that the proposed tag-based recommender system is a more cost effective approach compared to the coverage-based techniques.

VI. Threats to Validity

The main threat to validity is the choice of parameter values that we assigned to the number of tags, number of topics, and the interpolation parameter during the system tagging. Using different parameter values might produce different results. We also do not know how the results would change if we increased the size of the training datasets. To evaluate the quality of prioritization, we chose APFD, which is a commonly used measure in the field of regression test prioritization. APFD measures the quality of prioritized test cases based on how early the faulty test cases are positioned in the prioritized suite. However, APFD does not consider either the execution time of individual test cases or the severity of faults. Therefore, it can not measure the actual cost benefits that we are gaining from the prioritization. Another threat to validity is the generalization of our results. In this study, we used open-source applications, so the findings from our results cannot be interpreted in the context of industrial applications.

However, we tried to reduce this threat by using relatively large applications (15K to 312K LOCs) with large numbers of test cases (523 and 6,026) with real faults. Nevertheless, this threat can be further addressed by utilizing industrial software systems from various application domains.

VII. Related Work

Regression testing is a technique that practitioner apply to increase the reliability of a software by testing the system for validating program changes. To date, many techniques have been investigated by practitioners in which code coverage based technique are among the most commonly used techniques [60], [18], [27], [8], [62], [28], [51]. In this technique, the capability of a test case is measured by the amount of the code the test case can cover. The basic assumption is the test cases with higher code coverage have a better chance to reveal faults than test cases with lower code coverage. Although, previous studies show that these approaches can be effective in identifying regression faults, they suffers from the computational overhead and the demand for collecting and analyzing different test quality metrics (e.g. code coverage, fault history, etc.). Considering the increasing demands on rapid development and release cycles, the techniques that require expensive application costs would not be suitable for modern software development practices.

To address this limitation, recently some researchers have investigated automated techniques that can be aligned with modern software development environment. IR-based techniques are specific instances of these modern techniques. These techniques often receive a massive collection of documents and return the most relevant documents based on a given query. Because these techniques are effective, efficient, and easy to implement, they have been applied to a variety of software engineering problems. Bug localization, and traceability link recovery are two predominant software engineering tasks in which IR-based technique have been investigated [67], [11]. IR-based techniques also, have been used in other areas of software engineering such as software reusability [41], [48], change impact analysis in the source code [17], clone detection etc. [43].

Similar to our approach, there are researches that applied information retrieval to solve regression testing problems. In these techniques, a code commit is used as a query to search for relevant code/test from source code or other software artifacts. The output of these systems is a list of code documents that are relevant to the query. The relevance is measured by the textual similarity between the query and documents. IR-based RTP techniques are specific instances of IR-based feature location in source code and treaceability link recovery [22], [40], [42], which formulate regression testing as an information retrieval problem. What differentiates IR-based RTP from the common regression testing practices is the use of code changes as queries. These techniques often use additional information to improve the quality of retrieval. Additional information leveraged by existing IR-based RTP techniques includes: fault history [14], [44], [37], code structure [63], [59], [57], [10], textual diversity among tests [14], [55], [35], [36], etc.

Saha et al. [52] proposed a technique that maps the traditional regression testing problem to an information retrieval problem They used the differences between two program versions as a query and a test suite as a document, and then prioritized test cases based on their cosine similarities to the queries. Similarly, Azizi and Do [14] proposed an IR-based framework that maps test cases to a graph network and the retrieval algorithms searches through the network to find the most relevant test cases for a given query. Thomas [55] proposed a static black-box test prioritization approach using a topic model technique where test cases are ordered based on their edit-distances. This technique does not utilize any code change information and thus may be imprecise.

Although these proposed approaches show positive results in reducing the cost of regression testing, they suffer from low retrieval accuracy. One reason for the low accuracy of these techniques is that they merely utilize the test scripts as documents that are often short, and contain few non-discriminatory terms. The current state of research lacks methods to increase the quality of the documents in the regression testing domain. Our research seeks to apply tag-based search algorithm to generate a recommendation list for test prioritization. To our best knowledge, our tag-based recommender system for test prioritization is novel and has not yet been explored in regression testing.

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 program code change as a query and test files as document collection. It then annotates the test cases using *LDA* technique and stores the test cases in a database by assigning meaningful tags to the content of the tests. The output of our recommender system is a list of test cases that are highly likely exercising the portion of the program that has been changed. We evaluated our proposed approach using 37 versions from six programs with real regression faults and compared the results with five widely used prioritization techniques. The results of our empirical study indicate that our recommender system can improve test case prioritization, while reduces the cost of regression testing considerably. Because our proposed approach is built based on a text-retrieval and also eliminates coverage profiling, it can be aligned with today's agile software delivery practices in which traditional regression testing techniques might be less practical. For future work, we are interested to see whether it is beneficial to apply query reformulation techniques such as *Pseudo Relevance Feedback*. We hypothesize that the performance of IR-based regression test case prioritization can be improved by enhancing the quality of the queries.

References

[1] https://www.devart.com/codecompare/. [Accessed: Oct. 12, 2020].
[2] hhttp://www.nopcommerce.com/. [Accessed: Jan. 26, 2017].
[3] https://umbraco.com/. [Accessed: Oct. 06, 2017].

[4] https://github.com/rjust/defects4j. [Accessed: Oct. 25, 2020].

[5] https://github.com/nopSolutions/nopCommerce/issues//. [Accessed: Oct. 06, 2017].

[6] http://issues.umbraco.org/issues//. [Accessed: Oct. 06, 2017].

[7] http://www.eclemma.org/jacoco/trunk/doc/maven.html/. [Accessed: Aug. 06, 2020].

[8] K. K. Aggrawal, Yogesh Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. In *ACM SIGSOFT Software Engineering Notes*. ACM, 2004.

[9] Akiko Aizawa. An information-theoretic perspective of tf–idf measures. *Information Processing and Management*, 39:45–65, 2003.

[10] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol. Improving bug location using binary class relationships. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, page 174–183. ACM, 2012.

[11] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28, 2002.

[12] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. Software traceability with topic modeling. In *International Conference on Software Engineering (ICSE)*. ACM, 2010.

[13] Maral Azizi and Hyunsook Do. A collaborative filtering recommender systemfor test case prioritization in web applications. In *Symposium On Applied Computing*, page 107–114. ACM, 2018.

[14] Maral Azizi and Hyunsook Do. Retest: A cost effective test case selection technique for modern software development. In *International Symposium on Software Reliability Engineering*, pages 144–154. IEEE, 2018.

[15] David Binkley, Daniel Heinz, Dawn Lawrie, and Justin Overfelt. Understanding lda in source code analysis. In *International Conference on Program Comprehension (ICPC)*. ACM, 2014.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 28, 2003.

[17] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *IEEE International Software Metrics Symposium (METRICS)*, pages 144–154. IEEE, 2005.

[18] 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.

[19] Claudio Carpineto and Giovanni Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Computing Surveys*, January 2012.

[20] Jane Cleland-Huang, Patrick Mäder, Mehdi Mirakhorli, and Sorawit Amornborvornwong. Breaking the big-bang practice of traceability: Pushing timely trace recommendations to project stakeholders. In *International Requirements Engineering Conference (RE)*. IEEE, 2012.

[21] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Conference on Mining Software Repositories (MSR)*, pages 31–41. IEEE, 2010.

[22] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25:53–95, 2012.

[23] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, September 2006.

[24] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 113–124, November 2004.

[25] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering: An International Journal*, 11(1):33–70, January 2006.

[26] Frederico Durao and Peter Dolog. Extending a hybrid tag-based recommender system with personalization. In *Symposium On Applied Computing*. ACM, 2010.

[27] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. Technical Report 00-60-03, Oregon State University, February 2000.

[28] 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.

[29] Ido Guy, Naama Zwerdling, Inbal Ronen, David Carmel, and Erel Uziel. Social media recommendation based on people and tags. In *International Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 2010.

[30] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, and Andrea De Lucia. Evaluating the specificity of text retrieval queries to support software engineering. In *International Conference on Software Engineering (ICSE)*. ACM, 2012.

[31] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, and Andrea De Lucia. Automatic query performance assessment during the retrieval of software artifacts. In *International Conference on Automated Software Engineering (ASE)*. ACM, 2015.

[32] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *ACM Journal of Transactions on Software Engineering and Methodology*, 26, 2017.

[33] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *International Conference on Software Engineering (ICSE)*. ACM, 2013.

[34] Jiawei Han, Micheline Kamber, and Jian Pei. *Getting to Know Your Data*. elsevier, second edition, 2012.

[35] H. Hemmati, A. Arcuri, and L. C. Briand. Achieving scalable model based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, (1):1–43, 2013.

[36] H. Hemmati, Z. Fang, and M. V. Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[37] Yu-Chi Huang and Kuan-Li Peng nd Chin-Yu Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85:626–637, 2012.

[38] Dominik Kowald, Simone Kopeinik, and Elisabeth Lex. The tagrec framework as a toolkit for the development of tag-based recommender systems. In *International Conference on User Modeling, Adaptation and Personalization (UMAP)*. ACM, 2017.

[39] An Ngoc Lam, Anh Tuan Nguye, Hoan Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *International Conference on Program Comprehension (ICPC)*, pages 268–279. IEEE-ACM, 2017.

[40] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk. Information retrieval methods for automated traceability recovery. *Journal of Software and Systems Traceability*, 85:71–98, 2012.

[41] Yoelle Maarek, Daniel Berry, and Gail Kaiser. An information retrieval approach for automatically constructing software libraries. In *IEEE Software*. IEEE, 1990.

[42] A. Marcus and S. Haiduc. Text retrieval approaches for concept location in source code. In *Software Engineering: International Summer Schools, ISSSE*, page 126–158. Springer, 2013.

[43] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *In International Conference of Automated Software Engineering (ASE)*, page 107–114. IEEE, 2001.

[44] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing: An industrial case study. In *nternational Conference on Software Maintenance*, page 174–183. IEEE, 2013.

[45] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[46] Laura Moreno, Gabriele Bavota, Sonia Haiduc, and Massimiliano Di Penta. Query-based configuration of text retrieval solutions for software engineering tasks. In *International Conference on Foundation of Software Engineering (FSE)*. ACM, 2015.

[47] T. Bin Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *International Conference in Software Reliability Engineering (ISSRE)*. IEEE, 2015.

[48] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. In *IEEE Software*. IEEE, 1987.

[49] Mohammad Rahman and Chanchal Roy. Improving ir-based bug localization with context-aware query reformulation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 621–632. ACM, 2018.

[50] G. Rothermel, R. Untch, C. Chu, , and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE Inter-*

national Conference on Software Maintenance, pages 179–188. IEEE-ACM, 1999.

[51] 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.

[52] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *International Conference on Software Engineering (ICSE)*, pages 268–279. IEEE-ACM, 2015.

[53] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2012.

[54] Ahmed E. Hassan Dorothea Blostein Stephen W. Thomas, Bram Adams. Modeling the evolution of topics in source code histories. In *Working Conference on Mining Software Repositories (MSR)*. ACM, 2011.

[55] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic. In *Empirical Software Engineering*, pages 1–31. IEEE-ACM, 2012.

[56] Stephen W. Thomas. Mining software repositories using topic models. In *International Conference on Software Engineering (ICSE)*. ACM, 2011.

[57] Shuai Wang, David Buchmann, Shaukat Ali Simula, Arnaud Gotlieb, Dipesh Pradhan, and Marius Liaaen. Multi-objective test prioritization in software product line testing: An industrial case study. In *Proceedings of the International Software Product Line Conference* , pages 32–41, 2014.

[58] Song Wang, Jaechang Nam, and Lin Tan. Qtep: Quality-aware test case prioritization. In *Foundations of Software Engineerin (FSE)*, pages 523–535. ACM, 2017.

[59] Song Wang, Jaechang Nam, and Lin Tan. Qtep: Quality-aware test case prioritization. In *Foundations of Software Engineerin (FSE)*, pages 523–535. ACM, 2017.

[60] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[61] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 140–150, July 2007.

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

[63] K. C. Youm, J. Ahn, and E. Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.

[64] Valentina Zanardi and Licia Capra. Social ranking: Uncovering relevant content usingtag-based recomm ender systems. In *RecSys*. ACM, 2008.

[65] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42:530–543, 2016.

[66] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of ICSE*, pages 192–201. IEEE, 2013.

[67] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *International Conference on Software Engineering (ICSE)*, May 2012.

[68] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, PROMISE*. IEEE, 2007.