Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis

Lingchao Chen
The University of Texas at Dallas
lxc170330@utdallas.edu

Foyzul Hassan
The University of Texas at San Antonio
foyzul.hassan@my.utsa.edu

Xiaoyin Wang
The University of Texas at San Antonio
Xiaoyin.Wang@utsa.edu

Lingming Zhang
The University of Texas at Dallas
lingming.zhang@utdallas.edu

ABSTRACT

In modern software development, software libraries play a crucial role in reducing software development effort and improving software quality. However, at the same time, the asynchronous upgrades of software libraries and client software projects often result in incompatibilities between different versions of libraries and client projects. When libraries evolve, it is often very challenging for library developers to maintain the so-called backward compatibility and keep all their external behavior untouched, and behavioral backward incompatibilities (BBIs) may occur. In practice, the regression test suites of library projects often fail to detect all BBIs. Therefore, in this paper, we propose DeBBI to detect BBIs via *cross-project* testing and analysis, i.e., using the test suites of various client projects to detect library BBIs. Since executing all the possible client projects can be extremely time consuming, DeBBI transforms the problem of cross-project BBI detection into a traditional information retrieval (IR) problem to execute the client projects with higher probability to detect BBIs earlier. Furthermore, DeBBI considers project diversity and test relevance information for even faster BBI detection. The experimental results show that DeBBI can reduce the end-to-end testing time for detecting the first and average unique BBIs by 99.1% and 70.8% for JDK compared to naive cross-project BBI detection. Also, DeBBI has been applied to other popular 3rd-party libraries. To date, DeBBI has detected 97 BBI bugs with 19 already confirmed as previously unknown bugs.

ACM Reference Format:
Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380436

1 INTRODUCTION

As software products become larger and more complicated, library code plays an important role in almost any software. For example, while the sample Android app "Hello World" contains only several lines of source code, when it is executed on an Android mobile phone, it actually invokes libraries from the Android Software Development Kit (SDK), Java Development Kit (JDK), as well as the underlying Linux system. Third-party libraries such as Apache [14] and Square [13] libraries are also widely used in both open source and commercial software projects. The prevalent usage of software libraries has significantly reduced the software development cost and improved software quality.

At the same time, the asynchronous upgrades of software libraries and client software often result in incompatibilities between different library versions and client software. As techniques of computation evolve faster and faster, libraries are also upgraded more frequently, so do the occurrences of software incompatibilities. For example, Google releases a new major version of Android averagely every 11 months. After each major release, an outbreak of incompatibility-related bug reports will occur in GitHub, so do the version-upgrade-related negative reviews in the Google Play Market [56].

To avoid incompatibilities, for decades, "backward compatibility" has been well known as a major requirement in the upgrades of software libraries. However, in reality, "backward compatibility" is seldom fully achieved, even in widely used libraries. Some early research efforts (e.g., Chow and Notkin [31], Balaban et al. [26], and Dig and Johnson [33]) have confirmed the prevalence of backward incompatibility between two consecutive releases of software libraries. More recently, Cossette and Walker [32] identified 334 signature-level backward incompatibilities in 16 consecutive version pairs from 3 popular Java libraries: struts [5], log4j [12], and jDOM [11]. McDonnell et al. [56] identified 2,051 changes on method signatures in 13 consecutive Android API level pairs from API level 2-3 to API level 14-15. These studies all show that backward incompatibilities are prevalent. Furthermore, a recent study [58] found averagely over 12 test errors / failures from each version pair when performing cross-version testing on 68 consecutive version pairs of 15 popular Java libraries. This fact shows that, on top of signature-level backward incompatibilities, behavioral backward incompatibilities that may cause runtime errors instead of compilation errors are also prevalent.

Library incompatibilities may result in runtime failures both during the software development phase and after the software distribution. If the upgraded library is statically packaged in the client software product, the client developers may face some test failures when they try to incorporate the new release of the library. Thus they must perform extra changes and bug fixes if they want to take advantage of the new release of the library. In such a case, client developers may not be affected because they can still build the software product with the earlier library version. The case becomes worse when the upgraded library belongs to the runtime environment (e.g., operating system libraries, Java runtime libraries, platform libraries for plug-ins such as Chrome/-Firefox/Eclipse libraries). In such cases, a software user may simply perform a system/platform update (the user may even not notice it

if she turns on automatic updates) during the night, and suddenly find one or more software applications no longer working next morning. For example, Windows Vista is considered to be not very successful, and its failure has been largely ascribed to its backward incompatibility with Windows XP [1]. More recently, an upgrade of Android platform from 4.4 to 5.0 broke SougouInput, the most popular Chinese-input software with more than 200 million users [2]. Users could not input any Chinese character after they upgraded to Android 5.0, until a patch was released 4 days later.

This paper proposes to apply *cross-project* testing and analysis to overcome the challenges in BBI detection with the following two insights. *First, the large number of open source client software projects residing in open software repositories can serve as a natural knowledge base of common usage scenario and expected semantics of software library APIs. Second, it is difficult for natural language documents (e.g., release notes) to achieve comprehensiveness and preciseness in describing semantic changes of library APIs. In contrast, code (including library and client code, source and test code) can be better media to transfer knowledge from the library side to the client side.* In particular, to avoid BBI-related software runtime failures, to accelerate software upgrading process, and to reduce developer's effort in software migration, we propose DeBBI to detect BBIs on library side. Simple cross-version regression testing with built-in library test code may miss a lot of BBIs. For better detection of BBIs, DeBBI leverages the large number of existing client software projects in open software repositories, and performs large-scale testing on these projects with their built-in test code on the newer library version. Such largely expanded test suites may incur high costs. Therefore, we propose to transform the problem of cross-project BBI detection into a traditional information retrieval (IR) problem. More specifically, we treat the library-side API upgrades as the query, and the project-side usage of the library APIs as the document collection. Then, the projects with more intensively upgraded API uses will be prioritized for early execution to detect potential BBIs faster. Also, different projects may share similar API uses and thus detect similar BBIs. Thus, we further consider the diversity between client projects using the diversified Maximal Marginal Relevance (MMR) technique [30]. Finally, for each client project, we also optimize test executions by skipping the tests that may not touch the upgraded APIs. The paper makes the following contributions:

- **Idea.** We propose to solve the BBI detection problem via *cross-project* testing and analysis, and further transform the problem into a traditional IR problem.
- **Implementation.** We implement the proposed approach for testing library BBIs based on the ASM bytecode analysis framework [6] and the Indri IR framework [8].
- **Optimization.** We further propose to use MMR to consider the diversity of different client projects, and also extend traditional static regression test selection to the cross-project scenario to automatically skip the tests useless for BBI detection.
- **Study.** We present an extensive study on testing JDK and other popular 3rd-party library (such as Apache libraries) upgrades using tens of thousands of GitHub client Java projects. The experimental results show that DeBBI can reduce the end-to-end testing time for detecting the first and average BBI clusters by 99.1% and 70.8% for JDK, and detect 97 real BBI bugs (19 has been confirmed as previously unknown bugs).

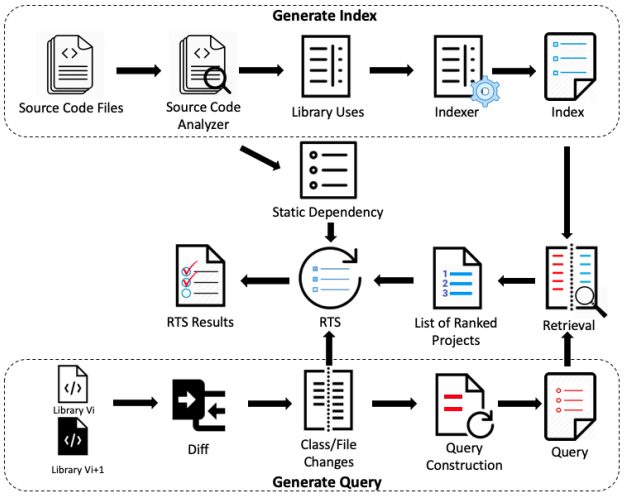

Figure 1: DeBBI structure

2 APPROACH

In this section, we first present the overview of our DeBBI approach (Section 2.1). Then, we illustrate how to apply IR techniques for efficient and effective BBI detection (Section 2.2). Finally, we present how to extend traditional Regression Test Selection (RTS) to the cross-project setting to further speed up DeBBI (Section 2.3).

2.1 Overview

Our DeBBI is a general approach for taming BBIs via cross-project testing, and can be applied to any library, including Android Software Development Kit (SDK) [4], Java Development Kit (JDK) [9], and third-party libraries such as Apache Software [14]. Figure 1 shows the overall architecture of our DeBBI. DeBBI takes two versions of the library under test and a set of client projects that directly use the library as input to find BBIs. DeBBI first extracts the changes (e.g., file changes) among the two library versions via static analysis. They are considered as queries in our IR model. Meanwhile, DeBBI preprocesses the source code for all the client projects to obtain the library APIs used by each project, and uses that to serve as the document for each project during IR. Then, DeBBI queries the library changes against the source code for all the client projects, so that the client projects accessing more changed APIs are tested earlier to detect BBIs faster.

Following prior work [53, 55, 71, 87], we performed stop word removal [40], stemming [62] for the IR document preparation. Note that we use all Java key words as our stop word since they are common for all Java projects. For each client project, we consider the class/file-level dependencies on the library under test as the document contents. For each class/file, we split its fully-qualified name into different words in the document or query. For example, we split `java.lang.String` into `java`, `lang` and `String`. These three words are all fed into our document or query. To ensure DeBBI effectiveness and efficiency, we further explore various IR

models in this work, including traditional and topic-model-based IR models (Details shown in Section 2.2). Furthermore, the client projects ranked high in the prioritization results may reveal similar or even the same BBIs. Therefore, we further consider the diversity of the IR results to detect *different* unique BBIs faster. To this end, we further use the Maximal Marginal Relevance (MMR) algorithm [30] to rank client projects with diverse library API uses.

IR models can help greatly reduce the number of client projects for finding BBIs. However, for each client project, all its tests are still executed. Therefore, in Section 2.3, we further use static analysis to compute the library APIs reachable from each test, and then compute the subset of tests which can potentially access changed library APIs as *affected tests*. In this way, for each client project, we only execute the affected tests to further speed up BBI detection.

2.2 DeBBI via Information Retrieval

Various IR models have been applied to solve software engineering problems, such as the Vector Space Model (VSM) [73], Latent Semantic Indexing (LSI) [47], and Latent Dirichlet Allocation (LDA) [27]. In theory, any IR model can be applied to DeBBI. In this work, we mainly consider two widely used IR models, VSM and LDA, due to their effectiveness [49, 83]. For each model, we studied state-of-the-art variants for effective BBI detection. Furthermore, for each studied variant, we further apply the Maximal Marginal Relevance (MMR) algorithm [30] to rank client projects with diverse library API uses.

2.2.1 Vector Space Model. Vector Space Model (VSM) [73] is an algebraic model for representing text documents and queries as vectors of indexed terms. TF.IDF (short for Term Frequency-Inverse Document Frequency) is a numerical statistic widely used to reflect word importance for a document under VSM. To date, TF.IDF and its variants (e.g., state-of-the-art Okapi BM25 [68]) have been widely recognized as robust and effective IR models [67]. Therefore, it has been widely studied and used in both IR and software engineering areas [60, 74, 81, 84]. Formally, assume that each document and query are represented by a term frequency vector $\vec{d}$ and $\vec{q}$ respectively, and n is the total number of terms or the size of vocabulary:

$$\vec{d} = (x_1, x_2, \ldots, x_n) \tag{1}$$

$$\vec{q} = (y_1, y_2, \ldots, y_n) \tag{2}$$

Element x_i and y_i are the frequency of term t_i in document $\vec{d}$ and query $\vec{q}$ respectively. Generally, query and document terms are weighted not just by their raw frequencies. There is a heuristic TF.IDF weighting formula to weight query and document term frequency (TF). Also, the inverse document frequency (IDF) is used to increase the weight of terms with low frequencies in the document and diminish the weight of terms which have high frequencies. Weighted vectors for $\vec{d}$ and $\vec{q}$ are computed as:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), \ldots, tf_d(x_n)idf(t_n)) \tag{3}$$

$$\vec{q_w} = (tf_d(y_1)idf(t_1), tf_d(y_2)idf(t_2), \ldots, tf_d(y_n)idf(t_n)) \tag{4}$$

Given a set D of source files for the client projects considered by DeBBI, the simplest and classic TF formulation just uses the raw count of each term in the document, i.e., the number of times that term t occurs in a document, which is given by $f_{t,d}$. Similarly, one simplest way to calculate IDF is given by $idf(t) = \log \frac{N}{n_t}$, where n_t is the number of documents with term t and N is the total number of documents in document collection D. Thus, one of the simplest ways to get TF.IDF score is to just multiply $f_{t,d}$ and $\log \frac{N}{n_t}$ to get term t's score in document $\vec{d}$, and then compute the vector similarity with query $\vec{q}$ to get document $\vec{d}$'s priority.

As we mentioned before, various TF.IDF variants have been proposed in practice. In this work, we use the Indri [8] framework, which includes various advanced algorithms to achieve more accurate models. The Indri's TF.IDF variant is based on Okapi BM25, which is a probabilistic retrieval framework model initially developed by Robertson et al. [68]. As to avoid division by zero, when a particular term appears in all documents, the IDF value here is: $idf(t) = \log \frac{N+1}{n_t+0.5}$. Meanwhile, the TF value is:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{len_d}{len_D})} \tag{5}$$

There are two tuning parameters $k1$ and b. $k1$ is used to calibrate document term frequency scaling. When $k1$ is just a small value, the term frequency value will quickly saturate; on the contrary, a large $k1$ value corresponds to using raw term frequency. $b(0 \leq b \leq 1)$ is used to determine the scaling by document length. When value b is 1, it corresponds to fully scaling the term weight by the document length, while $b = 0$ corresponds to no length scaling. Finally, len_d and len_D represent the current document length and average document length for the entire document collection, respectively.

Meanwhile, for the query's TF function, the length normalization is unnecessary because retrieval is applied with respect to a single fixed query. Therefore, we just set b as 0 here:

$$tf_q(y) = \frac{k_3 y}{x + k_3} \tag{6}$$

Thus, the similarity score of document $\vec{d}$ against query $\vec{q}$ is:

$$S(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \tag{7}$$

There are various configurations that we can choose in the Indri framework. One of them is the basic TF.IDF variant using BM25TF term weighting. It sets k_3 as 1000 in the equation 6. The only two parameters left for tuning are $k1$ (for term weight) and b (for term weight). We directly use their default values, i.e., 1.2 and 0.75, respectively. Another variant is Okapi, which performs retrieval via Okapi scoring. There are three parameters $k1$ (for term weight), b (for term weight), and $k3$ (for query term weight) in the variant. The default value of them are 1.2, 0.75 and 7 respectively. We also use these default values in our experiment. In this work, we use both models and denote them as TF.IDF and Okapi, respectively.

2.2.2 Latent Dirichlet Allocation. Different from VSM that directly represents documents with indexed terms, LDA further implements topic modeling in the retrieval process and computes generative statistical models to split a set of documents into corresponding topics with certain probabilities. In this way, each document is represented by the set of relevant abstract topics rather than the raw indexed terms. In the software engineering literature,

researchers have applied LDA to deal with bug localization [87], software categorization [80], or software repository analysis [76]. In those prior work, project source code is usually treated as LDA input documents. In contrast, in this work, DeBBI treats each client project's class-level dependency on the library under test as LDA input documents. Based on the input documents, LDA computes different topics for each of the client projects. The different topics indicate that there are different clusters of projects. When projects use very similar library APIs, they are assigned into similar topics.

Figure 2 shows the graphical model of LDA. The outer box D represents the documents. The inner box T represents the repeated choice of topics and words in a document. The generative process of model can be described as follows:

(1) Choose $T \sim \text{Poisson}(\epsilon)$
(2) Choose a topic vector $\theta \sim \text{Dir}(\alpha)$ for document D
(3) For each of the T terms w_i:
    (a) Choose a topic $z_j \sim \text{Multinomial}(\theta_D)$
    (b) Choose a term w_i from $p(w_i|z_j, \beta)$

For here, α is a smoothing parameter for document-topic distributions, and β is a smoothing parameter for topic-term distributions. The multinomial probability function p is:

$$p(\theta, z, w|\alpha, \beta) = p(\theta|\alpha) \prod_{n=1}^{T} p(z_n|\theta) p(w_n|z_n, \beta) \quad (8)$$

In this way, given a set of client projects, we first generate a term-by-document matrix $\vec{M}$. Then we use w_{ij} to represent the weight of i_{th} term in the j_{th} document. Note that following prior work [41, 45], we take TF.IDF as our weighting function, which can give more importance to words with high frequency in the current document and appearing in a small number of documents.

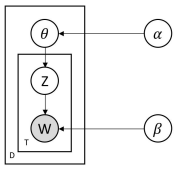

Figure 2: Graphical model for LDA

LDA further takes the $\vec{M}$ as input, and produces a topic-by-document matrix $\vec{R}$. For here, the probability that the j_{th} document belongs to the i_{th} topic is denoted by R_{ij} in this matrix. Because the number of topics is much smaller than the number of indexed terms in the corpus. LDA is mapping a high-dimensional space of documents into a low-dimensional space (represented using topics). The latent topics can be clustered by shared topics.

In the implementation, we apply the fast collapsed Gibbs sampling generative model [61] for LDA. The reason is that it is much faster and has the same accuracy compared against the standard LDA implementation [27]. There are the following parameters in the model which may affect its performance:

- t, which is the number of topics in the result. Follow the prior work [25], we set topic number as 10 in our experiment.
- n, which denotes the number of Gibbs iterations to train our model. And we set it as 10000 in the experiment following prior work [65].
- α, which influences the topic distributions per document. The topics will have a better smoothing effect when the α value is higher. We use the default value of 5.5.
- β, which influences the term's distribution per topic. The distribution of terms per topic will be more uniform with a higher β value. We use the default value of 0.01.

2.2.3 Maximal Marginal Relevance. Both the VSM and LDA techniques above will aggressively rank the most relevant client projects high in the list. However, the highly ranked projects may access similar library APIs and reveal the same BBIs repetitively. Therefore, in this work, we further consider the diversity among the search results to detect different unique BBIs faster. More specifically, we combine both VSM and LDA models with Maximal Marginal Relevance (MMR) [30] to solve this diversity issue to explore their performance. MMR has been widely studied in the IR community for diversified searching [38, 39, 43, 48]. Traditional IR models rank the retrieved documents in the descending order of relevance to the user's query. In contrast, MMR tries to measure relevance and novelty independently and consider them together via a linear combination to solve the diversity problem. For example, it maximizes marginal relevance in retrieval and summarization when a document is both relevant to the query and contains minimal similarity to the previously ranked documents. The MMR score equation can be formally defined as:

$$A_{rg} \max_{d_i \in D \setminus S} [\lambda(Sim_1(d_i, q) - (1-\lambda) \max_{d_j \in S} Sim_2(d_i, d_j))] \quad (9)$$

where D is the document collection (i.e., the set of considered client projects for testing a library using DeBBI) and q is the query (i.e., the changes among different library versions). S is the subset of documents which are already selected by IR. $D \setminus S$ is the set of not yet selected documents in D. Sim_1 and Sim_2 are the methods to measure similarity between documents and query. They can be the same or different. For here, we uniformly use BM25 [82] as our similarity calculation method. In the above definition, when parameter $\lambda = 1$, MMR gives us a standard relevance-ranked list. On the contrary, when $\lambda = 0$, MMR gives us a maximal diversity result. In addition, the sample information space is around the query when λ is a small number, whereas the larger value of λ will produce a result focusing on multiple potentially overlapping or reinforcing relevant documents. In our experiment we set λ as 0.5 which gives documents and queries the same weight.

2.3 Faster DeBBI via Testing Selection

Since the basic DeBBI only ranks client projects, all the tests within each tested projects still have to be executed. Therefore, we further extend DeBBI to reduce the number of test executions within each project. More specifically, we extend the traditional Regression Test selection (RTS) approach [69] to further enable even faster BBI detection. To date, various static and dynamic RTS techniques have been proposed in the literature [36, 37, 50, 75, 86]. In this work, we build DeBBI on top of state-of-the-art static RTS technique STARTS [50]. We chose STARTS since it has been demonstrated to be state-of-the-art static file-level RTS technique and can be competitive to state-of-the-art dynamic RTS technique Ekstazi [37]. Also, STARTS does not require prior dynamic execution information for each client project, which may not be available during BBI detection. STARTS is based on the traditional class firewall analysis firstly proposed by Leung et al. [46, 51]. To further consider the specific features of the Java programming language, STARTS

performs class firewall analysis on the Intertype Relation Graph (IRG) defined by Orso et al. [59]. The following presents the formal definition:

Definition 2.1 (Intertype Relation Graph). *The intertype relation graph of a given Java program can be formulated as a triple* $\langle \varepsilon, N_i, N_u \rangle$. *In the triple,* N *denotes the set of nodes representing all programs' classes or interfaces.* $\varepsilon_i \subseteq N \times N$ *denotes the set of inheritance edges. There exists an inheritance edge* $\langle n_1, n_2 \rangle \in \varepsilon_i$ *if type* n_1 *inherits from class* n_2*, or implements interface* n_2. $\varepsilon_u \subseteq N \times N$ *denotes the set of use edges. There exists an edge* $\langle n_1, n_2 \rangle \in \varepsilon_u$ *if type* n_1 *accesses any element of* n_2*, e.g., field references and method calls.*

There are two inputs for STARTS to select affected tests: (1) the set of changed files during software evolution, (2) the static dependency for each test computed based on the IRG graph, i.e., the files that can potentially be reachable from each test based on IRG. Then, STARTS computes all files that can potentially reach the changed files within the class firewall, and all tests within the firewall will be selected for execution. Formally, the class firewall can be computed as:

Definition 2.2 (Class Firewall). *The class firewall for a set of changed types* $\tau \subseteq N$ *is computed over the IRG* $\langle N, \varepsilon_i, \varepsilon_u \rangle$ *using as the transitive closure computation:* $firewall(\tau) = \tau \circ \overline{\mathcal{E}}^*$, *where* $\circ$ *is the relational product,* * *denotes the reflexive and transitive closure, and* $\overline{\mathcal{E}}$ *denotes the inverse of all use and inheritance edges, i.e.,* $(\varepsilon_i \cup \varepsilon_u)^{-1}$.

Note that the prior STARTS approach only analyzes the nodes within a project (ignoring all third-party and JDK libraries). On the contrary, in this work, we explicitly consider library changes, and aim to select the tests affected by library changes. Therefore, we augment the STARTS analysis to include library nodes. Note that (1) DeBBI only considers the nodes for the client projects and the library under test, and ignores all the other library nodes, and (2) DeBBI only considers the library nodes directly reachable from client projects. The reason is that the nodes for other libraries are not of interest, and the library nodes not directly reachable from the client projects may not have clear impact on the current project. For example, when applying DeBBI to detect JDK BBIs, we don't consider the third-party library dependencies and only collect the source code and test code JDK dependencies through jDeps [10]. Then we set the changed JDK library files as our code changes for test selection. Note that, we further filter out the top 200 most widely used JDK files, such as `java.lang.String` and `java.util.List`. The reason is that these files are almost used by all projects/tests and cannot help much in test selection. Note that we empirically validated that after filtering these JDK classes, our test selection is still safe, i.e., not missing any unique BBI.

Figure 3 illustrates how we adapt RTS for detecting BBIs for JDK. In the example IRG, the inheritance and use edges are marked with label "*i*" and "*u*". L denotes a third-party library node, which uses JDK node JDK_5; C is a client project node which inherits library L and uses JDK_1 and JDK_2. There are three tests T_1, T_2 and T_3 all using JDK_3. According to our approach, we do not consider the dependencies of third-party library, and thus JDK_5 will not be considered in our dependency result (pruned by red cross mark). In addition, we just consider one layer JDK dependency. For example, we only collect JDK dependencies of C, T_1, T_2 and T_3. We do not consider the further dependencies of JDK_1, JDK_2, JDK_3 and JDK_4. From the figure, T_2 uses client C and T_3 uses JDK_4, respectively. JDK_1, JDK_3 and JDK_4 are the changed JDK classes (marked with gray shadow). Note that JDK_3 is one of the 200 most commonly used JDK class, and it will not be considered in JDK diff results as discussed before (marked with dashed oval). In this way, T_2 can potentially reach JDK_1 and T_3 is using changed class JDK_4. Thus, T_2 and T_3 are affected tests in our RTS technique, marked within the dashed area (i.e., our class firewall).

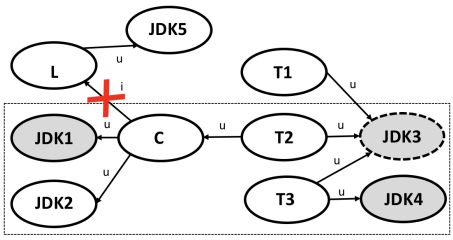

Figure 3: Example IRG

Table 1: Dataset summary

Description	Min	Max	Avg.
# Number of Java Files per Project	1	12979	130.37
# Number of Test Cases per Project	0	665028	329.68

3 EXPERIMENTS AND ANALYSIS

In this section, we first described our dataset for detecting JDK BBIs (Section 3.1), followed by our evaluation environment (Section 3.2), evaluation metrics (Section 3.3), research questions (Section 3.4), and results (Section 3.5). Finally, we discuss the threats to validity in Section 3.6.

3.1 Dataset

To construct the dataset for detecting JDK BBIs, we first collect all the most-forked Java projects with over 20 forks from the GitHub repository. It returns a collection of 8,481 unique Java projects. In these resulting projects, 4,928 of them support the Maven build system. Finally, we use all the 2,953 remaining projects can pass the build and test phases successfully under JDK 8 as the dataset for this study.

Table 1 describes the dataset in more details. In particular, the number of Java source files in a project ranges from 1 to 12,979, and the number of test cases in a project ranges from 0 to 665,028. The average number of Java source files and the average number of test cases are 130.37 and 329.68, respectively. Since we would like to find BBI issues for different versions of JDK, the same dataset is applied to build and test with different JDK versions.

3.2 Experiment Settings

To perform our experiment, we need a set of confirmed JDK BBI bugs as ground truth. We use the dataset described in Section 3.1 to detect such confirmed BBI bugs. The intuition is that, we can confirm a BBI bug by checking whether it is fixed in the later versions of JDK. If a test case passes in JDK 8 but fails in JDK 9.0.0, then it reveals a BBI between JDK 8 and 9.0.0. However, we are not sure whether this BBI is an intended behavior change by JDK developers or a BBI bug. To confirm that such a BBI is a BBI bug,

we further run the test case on 9.0.1, and if the BBI disappears, we confirm that the test failure in JDK 9 reveals a BBI bug. To categorize duplicated BBI bugs, we manually cluster all the reported BBIs caused by the same root issues to identify unique BBI bugs. In this way, we define every reported BBI as a *raw BBI bug* and every clustered BBI as a *unique BBI bug*. Note that we consider both raw and clustered bugs to better measure DeBBI effectiveness.

When performing the build and testing, we use Maven 3.3.9 to build and test each project. For the JDK version, We use JDK 8.0.161, 9.0.0 and 9.0.1. We use a computer with Intel(R) Xeon(R) CPU 2.60GHz with 528GB of Memory, and Ubuntu 16.04.3 LTS operating system.

3.3 Evaluation Metrics

We use each of the following three metrics to evaluate the number of projects tested, the number of test executions and time taken to identify BBIs:

- **First:** This metric reports the number of client software projects tested, the number of tests executed, or time (in second) taken to identify the first BBI bug. This metric emphasizes fast detection of the first BBI, which is essential for the developers to start debugging earlier.
- **Average:** This metric is the average number of client software projects tested, tests executed, or average time taken to find each BBI. This metric emphasizes fast detection of BBIs in average cases.
- **Last:** Like the **First** metric, this metric reports the number of client software projects evaluated, the number of tests executed and time taken to identify the last BBI. This metric emphasizes fast detection of all BBIs.

3.4 Research Questions

We seek to answer following four research questions:

- **RQ1:** Is DeBBI more effective than random project prioritization in identifying BBI issues?
- **RQ2:** How does diversity resolution technique help improve the performance of DeBBI?
- **RQ3:** Can we further boost DeBBI via extending traditional static Regression Test Selection (RTS)?
- **RQ4:** How does DeBBI perform in case of parallel execution?
- **RQ5:** Can DeBBI be generalized to other popular 3rd-party libraries besides JDK?

3.5 Results

RQ1: Basic DeBBI vs. Random Project Prioritization. To evaluate DeBBI on detecting BBIs for JDK, we compared the basic IR-based DeBBI with the Random technique, which randomly sorts client projects to identify BBIs. Also, the Random technique results are averaged over 5 runs to isolate the impact of random factors. We compared our results with the Random technique from three aspects: i) effectiveness in the number of tested client software projects, ii) effectiveness in the number of executed tests, and iii) effectiveness in test execution time. For each aspect, we measure the **First**, **Average**, and **Last** metrics of both the Random and our IR-based techniques. The results are presented in Table 2. In the left half of the table, we present the **First**, **Last**, and **Average** values on client software projects, test executions, and execution time without bug clustering. The values in the bracket are the relative reduction for the corresponding metrics compared with the Random technique. The best technique for each metric has also been marked in gray.

We have following observations for the bugs without clustering: First, all IR-based techniques perform much better than Random technique on the **First** values, with mostly 60% to 90% reduction on all three aspects. However, if we consider **Average** and **Last** values, the enhancement of IR-based techniques is not that significant, especially for execution time. This can be due to the lack of diversity in IR-based prioritization results. Second, there is none IR-based technique that outperforms all other techniques, but LDA is performing better (with 4.7% to 82.3% reduction) than Random technique on all values from all aspects.

As same BBI bugs can appear in multiple projects and test cases, we also performed BBI clustering to check how different techniques compare on identifying different *unique* BBI bugs. The right half of Table 2 shows the effectiveness of IR based techniques and Random technique on unique BBI bugs. The data presentation is the same as the left half. We have similar observations compared with left half of the table: IR-based techniques perform much better on **First** values, but not so good on **Last** and **Average** values. Furthermore, in general, IR-based techniques perform better than the Random technique on all values in test execution time for unique BBI bugs. The reason is that for unique BBI bugs DeBBI only need to find the first raw BBI bug in each cluster, making it easier for IR-based DeBBI to find unique BBI bugs faster.

RQ2: Diversity Enhancement. To check whether diversity enhancement techniques such as Maximal Marginal Relevance (MMR) can enhance IR-based project prioritization, we combine MMR with all IR-based techniques TF.IDF, Okapi and LDA. Table 3 shows the effectiveness of MMR-integrated IR-based techniques. From the table, we can see that although MMR is not very helpful on some IR techniques (TF.IDF and Okapi) in all aspects, it is able to enhance the LDA-based technique significantly. LDA+MMR outperforms all other techniques on almost all values from all aspects. Comparing with results in Table 2, we can see that MMR technique can enhance LDA-based technique on five of nine evaluated metrics without bug clustering and seven of nine metrics with bug clustering. In particular, when it comes to with bug clustering, LDA+MMR is able to reduce 99.1%, 59.0%, and 65.4% of test execution time to detect the **First**, **Last**, and **Average** unique BBI bugs, which is a huge enhancement over the Random technique.

RQ3: Static Regression Test Selection (RTS). When a library gets updated, not all the tests from its client projects are affected by the library code changes. If we can remove such irrelevant test cases, we may further enhance the reduction on the number of test executions and execution time. Therefore, we further exclude the test cases that will not be affected by JDK code changes via RTS. The results of techniques with RTS combined are presented in Table 4, where the Random technique is used as the baseline for comparison. From the table, we can see that, with RTS combined, even Random+RTS also achieves good effectiveness (average execution time reduced from more than 70K seconds to about 41K seconds); meanwhile, DeBBI models tend to have even larger improvements. In addition, on detecting clustered unique BBI bugs, the LDA+MMR

Table 2: Effectiveness of the basic DeBBI

	Without Bug Clustering									With Bug Clustering								
	Client Software Projects			Test Case			Execution Time(sec)			Client Software Projects			Test Case			Execution Time(sec)		
	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average
Random	63	2702	1607	1253	961050	776892	1494.97	109005.21	73360.5	63	2402	1663	1253	947219	758567	1494.97	103965.45	76463.23
TF.IDF	**3 (95.2%)**	2487 (8.0%)	**1322 (17.7%)**	**32 (97.4%)**	964053 (-0.3%)	399301 (48.6%)	**53.9 (96.4%)**	110727.5 (-1.6%)	77340.8 (-5.4%)	**3 (95.2%)**	1901 (20.9%)	1135 (31.7%)	**32 (97.4%)**	948943 (-0.2%)	413608 (45.5%)	**53.9 (96.4%)**	88637.3 (14.7%)	67582.5 (11.6%)
Okapi	5 (92.1%)	**2379 (12.0%)**	1375 (14.4%)	48 (96.2%)	962737 (-0.2%)	457375 (41.1%)	91 (93.9%)	109132.7 (-0.1%)	74956.7 (-2.2%)	5 (92.1%)	**1888 (21.4%)**	**982 (41.0%)**	48 (96.2%)	949122 (-0.2%)	241894 (68.1%)	91 (93.9%)	**87215.6 (16.1%)**	**60150.7 (21.3%)**
LDA	43 (31.7%)	2445 (9.5%)	1532 (4.7%)	573 (54.3%)	**727141 (24.3%)**	**167110 (78.5%)**	263.9 (82.3%)	**94113.1 (13.7%)**	**48290.1 (34.2%)**	43 (31.7%)	2332 (2.9%)	1747 (-5.1%)	573 (54.3%)	**711989 (24.8%)**	**108083 (85.8%)**	263.9 (82.3%)	90799.8 (12.7%)	64822.2 (15.2%)

Table 3: Effectiveness of DeBBI with MMR

	Without Bug Clustering									With Bug Clustering								
	Client Software Projects			Test Case			Execution Time(sec)			Client Software Projects			Test Case			Execution Time(sec)		
	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average
Random	63	2702	1607	1253	961050	776892	1494.9	109005.2	73360.5	63	2402	1663	1253	947219	758567	1494.9	103965.5	76463.2
TF.IDF+MMR	28 (55.6%)	2404 (11.0%)	1306 (18.7%)	5791 (-362.2%)	961604 (-0.1%)	**515462 (33.7%)**	4104.5 (-174.6%)	109603.5 (-0.5%)	79052.8 (-7.8%)	28 (55.6%)	1591 (33.8%)	867 (47.9%)	5791 (-362.2%)	944968 (0.2%)	**369206 (51.3%)**	4104.5 (-174.6%)	85428.7 (17.8%)	59618.8 (22.0%)
Okapi+MMR	25 (60.3%)	2398 (11.3%)	1324 (17.6%)	5759 (-359.6%)	963338 (-0.2%)	559859 (27.9%)	4057.2 (-171.4%)	109540.7 (-0.5%)	81400.4 (-11.0%)	25 (60.3%)	1672 (30.4%)	878 (47.2%)	5759 (-359.6%)	949900 (-0.3%)	450257 (40.6%)	4057.2 (-171.4%)	86264.9 (17.0%)	59906.6 (21.7%)
LDA+MMR	**1 (98.4%)**	**2340 (13.4%)**	**1243 (22.7%)**	**1 (99.9%)**	**959254 (0.2%)**	759536 (2.2%)	**12.7 (99.1%)**	**105832.7 (2.9%)**	**55970.2 (23.7%)**	**1 (98.4%)**	**1029 (57.2%)**	**616 (63.0%)**	**1 (99.9%)**	**931735 (1.6%)**	553400 (27.0%)	**12.7 (99.1%)**	**42645.6 (59.0%)**	**26433.9 (65.4%)**

technique, which has achieved best effectiveness without RTS, still achieves significant enhancement over the Random technique when RTS is combined. Specifically, LDA+RTS can achieve 63.2% reduction on detecting raw BBI bugs and LDA+MMR+RTS can achieve 70.8% reduction on detecting unique BBI bugs compared with the Random technique on **Average** execution time. In other words, DeBBI can save 1017.1 hours to find all raw BBI bugs and 120.4 hours to find all unique BBI bugs.

In reality, detecting a new unique BBI bug is apparently more important than finding another instance of a known BBI bug. Therefore, we believe LDA+MMR+RTS is the best technique that we recommend to be used by default in reality. To make it more convenient to check the necessity of each used component (i.e., LDA, MMR, and RTS) compared to baseline techniques, we present the comparison among four selected techniques: Random technique, LDA, LDA+MMR, and LDA+MMR+RTS on clustered unique BBIs in Figures 4 to 6.

In particular, Figure 4 compares all four techniques on their **First**, **Last**, and **Average** values on the number of client project executions. Figure 5 and Figure 6 present similar comparison on the number of test executions and execution time. As shown in Figure 4, for prioritization of the client software projects, since RTS does not optimize project selection, LDA+MMR and LDA+MMR+RTS show same effectiveness. However, if we compare LDA+MMR+RTS with Random approach, it shows 98.4% 57.2% and 63.0% reduction on **First**, **Last**, and **Average** values respectively over the Random technique. As shown in Figure 5, from the aspect of test cases, LDA+MMR+RTS achieves 99.9%, 97.6%, and 97.6% for **First**, **Last**, and **Average** values over Random technique. As shown in Figure 6, from the aspect of execution time, LDA+MMR+RTS achieves 99.1%, 68.0%, and 70.8% reduction **First**, **Last**, and **Average** values over Random technique.

RQ4: DeBBI Effectiveness for Parallel Execution. We further utilized the `multiprocessing` package of Python for parallel project execution. We used Python `Pool` to control the different processes to start or join in the main process and used `Manager` and `Queue` to control the shared resource between processes. In our experiments, the ranked project list from our IR-based result is the shared resource. Sub-processes try to get the project from queue and run it. As soon as one process finishes execution, it starts to get the next one to run. Here, we use 5 sub processes in our experiment to evaluate our technique. Table 5 shows the results of DeBBI with and without bug clustering during parallel execution. The left part is the execution time without bug clustering and right part is the execution time with bug clustering. Column 1 list all techniques. Columns 2-7 list **First**, **Last** and **Average** value of execution time to find raw BBI bugs and unique BBI bugs respectively. We use the Random technique with multiprocessing as our baseline technique. From the results, we can see that TF.IDF with MMR, Okapi and LDA with MMR all can find first raw BBI bug and unique BBI bug in 12.7 seconds with the 84.7% reduction compared to Random. LDA has the best performance in **Last** and **Average** with 11.8 % and 38.4 % reduction without bug clustering. Meanwhile, TF.IDF with MMR has the best performance in **Last** and **Average** with 80.9 % and 63.2 % reduction with bug clustering.

Table 6 shows the results when combining our IR-based techniques with RTS during parallel project execution. We still use the Random technique with multiprocessing as our baseline to check the results. From the results, all techniques combined with RTS can have a huge enhancement in **Last** and **Average** value of execution time. The reason LDA+RTS is better than Random in **First** is that RTS does not have too much help here. Random and most techniques can find first bug fast without RTS and executing RTS needs extra overhead[1]. Thus, the performance of **First** is not very good here. However, LDA+MMR+RTS is able to have 71.4 % and 83.1 % reduction in **Last** without and with bug clustering. LDA+RTS can have 64.4 % and 60.8 % average time reduction to find raw BBI bugs and unique BBI bugs. To sum up, LDA+MMR+RTS is still one of the most effective techniques in the setting of parallel project execution. It can save 129.3 hours to find all raw BBI bugs and 9.9 hours to find all unique BBI bugs compared to Random technique with parallel execution.

RQ5: DeBBI Application to Other Libraries. Besides JDK, we further use other popular libraries to thoroughly evaluate the performance of our approach. For this experiment, we cloned all Maven-based Java projects that are created between August 2008 and December 2019 on GitHub with at least one star, and finally included 56,092 unique projects that can successfully pass the build and test phases in our dataset. In total, there are 40,191 3rd-party libraries used by the projects in our client project dataset. We then

[1]Note that all the RTS overhead costs, including computing dependencies and performing RTS analysis, are considered in the our DeBBI time measurement.

Table 4: Effectiveness of DeBBI with RTS

	Without Bug Clustering						With Bug Clustering					
	Test Case			Execution Time(sec)			Test Case			Execution Time(sec)		
	First	Last	Average	First	Last	Average	First	Last	Average	First	Last	Average
Random	1253	961050	776892	1494.9	109005.2	73360.5	1253	947219	758567	1494.9	103965.5	76463.2
Random+RTS	337 (73.1%)	27016 (97.2%)	14402 (98.1%)	1257.5 (15.9%)	71083.7 (34.8%)	40856.8 (44.3%)	337 (73.1%)	23985 (97.5%)	15039 (98.0%)	1257.5 (15.9%)	62587.8 (39.8%)	41810.2 (45.3%)
TF.IDF+RTS	6 (99.5%)	28013 (97.1%)	21564 (97.2%)	303.6 (79.7%)	74613.8 (31.6%)	52714.6 (28.1%)	6 (99.5%)	27021 (97.1%)	18918 (97.5%)	303.7 (79.7%)	67659.2 (34.9%)	46428.2 (39.3%)
TF.IDF+MMR+RTS	1474 (-17.6%)	27987 (97.1%)	23073 (97.0%)	3260.2 (-118.1%)	74642.2 (31.5%)	54315.2 (26.0%)	1474 (-17.6%)	26298 (97.2%)	18922 (97.5%)	3260.2 (-118.1%)	63198.1 (39.2%)	41769.4 (45.4%)
Okapi+RTS	2 (99.8%)	27719 (97.1%)	22009 (97.2%)	**82.9 (94.5%)**	98866.6 (9.3%)	76910.7 (-4.8%)	2 (99.8%)	26787 (97.2%)	17881 (97.6%)	278.3 (81.4%)	66228.2 (36.3%)	43050.3 (43.7%)
Okapi+MMR+RTS	739 (41.0%)	27996 (97.1%)	23457 (97.0%)	3038.8 (-103.3%)	74678.5 (31.5%)	55316.7 (24.6%)	739 (41.0%)	26698 (97.2%)	18636 (97.5%)	3038.8 (-103.3%)	64302.9 (- 38.1%)	42449.7 (44.5%)
LDA+RTS	210 (83.2%)	**9284 (99.0%)**	**4020 (99.5%)**	507.3 (66.1%)	**50285.1 (53.9%)**	**27010.3 (63.2%)**	210 (83.2%)	**7535 (99.2%)**	**4221 (99.4%)**	507.3 (66.1%)	46847.2 (54.9%)	31159.9 (59.2%)
LDA+MMR+RTS	**1 (99.9%)**	26287 (97.3%)	22274 (97.1%)	197 (86.8%)	69353.1 (36.4%)	46072.8 (37.2%)	**1 (99.9%)**	22692 (97.6%)	18003 (97.7%)	**12.7 (99.1%)**	**33241.9 (68.0%)**	**22300.2 (70.8%)**

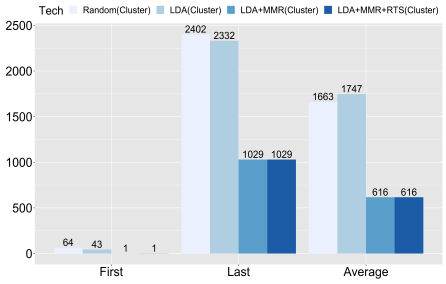

Figure 4: Client project execution

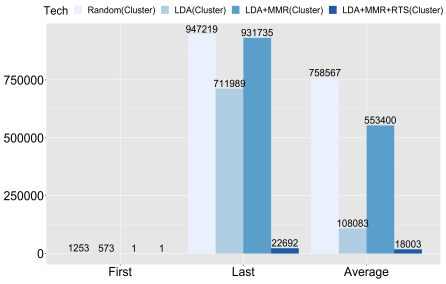

Figure 5: Test case execution

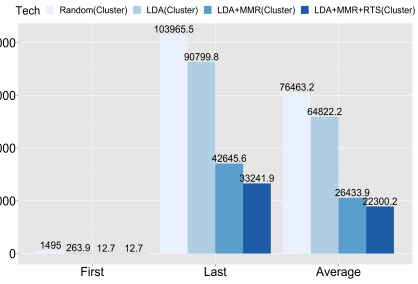

Figure 6: Test execution time

Table 5: DeBBI for parallel project execution

	Without Bug Clustering			With Bug Clustering		
	Execution Time(sec)			Execution Time(sec)		
	First	Last	Average	First	Last	Average
Random	83.5	53122.1	15026.5	83.5	51898.8	7695.1
TF.IDF	53.9 (35.2%)	68041 (-28.1%)	15916.4 (-5.9%)	53.9 (35.2%)	42494.4 (18.1%)	9024.3 (-17.3%)
TF.IDF+MMR	**12.7 (84.7%)**	49746.5 (6.4%)	15634 (-4.0%)	**12.7 (84.7%)**	**9905.3 (80.9%)**	**2832.8 (63.2%)**
Okapi	**12.7 (84.7%)**	51973.8 (2.2%)	15536.1 (-3.4%)	**12.7 (84.7%)**	50407.9 (2.9%)	7483.5 (2.7%)
Okapi+MMR	548 (-558.9%)	52171.3 (1.8%)	16080.9 (-7.0%)	548 (-558.9%)	49159.6 (5.3%)	8037.3 (-4.4%)
LDA	57 (31.4%)	**46837.4 (11.8%)**	**9262.9 (38.4%)**	57 (31.4%)	46180.8 (11.0%)	6948.8 (9.7%)
LDA+MMR	**12.7 (84.7%)**	47886.4 (9.9%)	10530.2 (29.9%)	**12.7 (84.7%)**	18754.7 (63.9%)	3386.2 (56.0%)

Table 6: DeBBI with RTS for parallel project execution

	Without Bug Clustering			With Bug Clustering		
	Execution Time(sec)			Execution Time(sec)		
	First	Last	Average	First	Last	Average
Random	83.5	53122.1	15026.5	83.5	51898.8	7695.1
Random+RTS	150.9 (-80.7%)	19495 (63.3%)	8072.8 (46.3%)	150.9 (-80.7%)	17712.4 (65.9%)	4477.7 (41.8%)
TF.IDF+RTS	303.6 (-263.6%)	21391.5 (59.7%)	10575.2 (29.6%)	303.6 (-263.6%)	15991.7 (69.2%)	5299.4 (31.1%)
TF.IDF+MMR+RTS	197 (-135.9%)	18115.2 (65.9%)	10785.8 (28.2%)	197 (-135.9%)	12176.8 (76.5%)	4142.7 (46.2%)
Okapi+RTS	197 (-13.6%)	17437 (67.2%)	10656 (29.1%)	197 (-13.6%)	14250.2 (72.5%)	5019.6 (34.8%)
Okapi+MMR+RTS	889 (-964.7%)	20250.1 (61.9%)	11076.6 (26.3%)	889 (-964.7%)	16087.6 (69.0%)	5593.4 (27.3%)
LDA+RTS	**95.7 (36.6%)**	17016.2 (68.0%)	**5352.6 (64.4%)**	**95.7 (36.6%)**	11141 (78.5%)	**3014.7 (60.8%)**
LDA+MMR+RTS	197 (-135.9%)	**15180.3 (71.4%)**	9135.5 (39.2%)	197 (-135.9%)	**8757.4 (83.1%)**	3257.4 (57.7%)

sort all libraries by use frequency and randomly choose 100 libraries from the top 300 to detect BBI bugs through DeBBI.

During our manual inspection, we found there are three types of false positives reported by DeBBI: (1) failures triggered by Maven POM file specifications (e.g., the specific updated library versions are prohibited by `POM.xml`), (2) failures triggered by intended changes (e.g., due to deprecated methods/implementations), and (3) failures triggered by dependency conflicts (e.g., the library updates are not compatible with specific versions of other libraries). Types (1) and (2) have their corresponding specific stack traces with fixed patterns. Thus, we were able to develop a rule-based method in DeBBI to automatically filter out them. However, we cannot avoid the false positives from Type (3). After manually removing 22 Type (3) false positives, DeBBI reported 97 unique BBI bugs. To date, 19 bugs have been confirmed as previously unknown bugs. 54 bug has been confirmed as previously known bugs (e.g., for COLLECTIONS-721 [16]), while all the other bug reports are still under active discussion. Interestingly, among the bug reports still under discussion, some reports have already been confirmed by other users (e.g. *"Experiencing same issue."* for reflection-277 [23]) even though not yet confirmed by the actual library developers.

Quantitative analysis. Due to the space limitation, we only present partial experimental results for the library projects with confirmed previously unknown BBI bugs in Table 7. In the table, Columns 1-4 list all the libraries, the number of corresponding GitHub Stars, the number of client projects from our dataset using the corresponding libraries, and the revision ranges that we use to detect BBIs. Columns 5-7 further present the number of unique unknown, known, and under discussion BBI bugs reported by DeBBI for this subset of libraries. Columns 8-13 present the **First**, **Last**, and **Average** values in terms of the number of test executions and execution time for our default LDA+MMR+RTS technique (with improvement over the Random technique shown in the parenthesis). The experiment parameters used are the same as our JDK experiment. From the table, we can observe that DeBBI can consistently improve the

```java
@Test
public void demo() {
    Map<Path, String> test = new HashMap<Path, String>();
    Path path = Paths.get("/tmp/test/file");
    test.put(path, "pathMD5");
    assertThat(test)
    .containsOnlyKeys(path)
    .containsValue("pathMD5");
}
```

Figure 7: Assertj-core-1751 [15] triggering test

```java
@Test
public void demo() throws FileSystemException{
    final String URI = "maprfs:///";
    UrlFileNameParser parser = new UrlFileNameParser();
    FileName name = parser.parseUri(null, null, URI);
    assertEquals(URI, name.getURI());
}
```

Figure 8: Commons-vfs-739 [24] triggering test

BBI detection efficiency in all traced metrics, further demonstrating the effectiveness of DeBBI.

Qualitative analysis. For the 19 confirmed previously unknown BBI bugs, developers quickly fixed the buggy code for 4 of them, and even added our reported test case in their regression test suites for 3 of them. For example, Figure 7 shows the test for issue Assertj-core-1751 [15]. Method `containsOnlyKeys` cannot handle the case when the `containsOnlyKeys` API is invoked on a `Map` with key type `Path`. This test is challenging to generate automatically due to the special corner case, while DeBBI is able to directly obtain such tests for free from client projects, demonstrating the promising future of DeBBI. Interestingly, at first one developer found it too difficult to fix it and wanted to just add a breaking-change notice; later on, another developer proposed a solution to finally fix it. Issue Commons-vfs-739 [24] is triggered when using Apache Commons-vfs to parse a MapR File System file path (shown in Figure 8). It is also challenging to generate this test automatically since the bug will be triggered only when the first two parameters for method `parseUri` are both `null` and `URI` includes the substring ":///". Furthermore, issue Jsoup-1274 [20] from library `Jsoup`, a widely used Java HTML parser, is incurred by the change of the method `select` – the developers forgot to deal with the situation when the end of the string in method `select` is a space (shown in Figure 9). The method `select` should trim the space first and continue to parse the string, but it throws an exception. DeBBI is able to detect it through a special test case that used `Jsoup` to parse a specific string followed by a whitespace. The developers were also quite active in fixing issue mybatis-spring-427 [22] reported by DeBBI, saying: *"Thanks for your report! This issue is bug(This issue was included by 5ca5f2d). We will revert it at 2.0.4."*

11 other confirmed BBI bugs are mitigated by the developers via changing the documents, since the developers did not realize they were BBI bugs until we submitted the reports and could not undo the change or fix the code. These BBI bugs were mitigated by adding an announcement in the corresponding documents. For example, the following comment is from the issue java-jwt-376 [18]:

"You are correct that this would be a breaking change, so should have been targeted at a future major version or at the very least called out the breaking change in the CHANGELOG.md file. Unfortunately, at this point we cannot undo the change without breaking others who are not handling the UnsupportedEncodingException. We should update the Change log, so keeping this issue open to address that. Apologies for the inconvenience, and thank you for raising this."

```java
@Test
public void demo(){
    String content = "<p> Select Test";
    StringBuilder bodyHtml = new StringBuilder();
    bodyHtml.append(content);
    Document document = Jsoup.parse(bodyHtml.toString());
    StringBuilder nav = new StringBuilder();
    Elements bodyElements = document.select("body > * ");
}
```

Figure 9: Jsoup-1274 [20] triggering test

For the remaining 4 confirmed BBI bugs, issues lombok-2320 [21] and HttpAsyncClient-159 [17] cannot be easily fixed by the developers for the moment. For example, the Apache HttpAsyncClient developers said:

"There is no much we can do about it now. If we remove the offending constructor to restore full compatibility with 4.1.3 we will break full compatibility with 4.1.4."

The other 2 unfixed bugs are from Apache Commons-io and Apache Jena. They confirmed our reported BBI bugs are source incompatibility, but cannot afford to fix them. For example, the Apache Jena [19] developers said:

"We try to migrate gracefully, and it is a compile time error. There is a balance between compatibility and building up technical debt. Change away from use of FastDateFormat was forced on the code (staying at the old version forever is not an option). Sometimes, our understanding of what users do, and do not use, is incomplete. "

3.6 Threats to Validity

The major internal threat to our evaluation is whether our ground truth on incompatibility bugs is correct. For JDK, although large-scale client testing reveals a lot of test failures, their causes are different and may not always indicate incompatibilities of JDK. For example, Raemaekers et al. [64] observed that library-breaking changes have a huge impact on project compilation. To reduce this threat, we use the test failures that are fixed when using Java 9.0.1 as the ground truth because they are incompatibility issues confirmed by JDK developers. This solution is not perfect as we may miss some real JDK incompatibilities and bugs that are not noticed and confirmed by JDK developers. For the popular 3rd-party libraries, we manually inspected all the reported cases (since they are more affordable than the JDK experiments) to confirm the ground truth, and also filed corresponding bug reports for the software developers to confirm. The major external threat to our evaluation is whether our approach may be generalized to libraries other than the studied ones. It should be noted that JDK is not a single library but a collection of tens of Java packages and even libraries developed by the 3rd-party such as SAXP libraries by XML-DEV and DOM libraries by W3C. To reduce such threats, we have also applied DeBBI to detect BBIs for other widely used 3rd-party libraries from GitHub. In the future, we further plan to further apply our DeBBI to other widely-used libraries such as the Android SDK.

Table 7: Effectiveness of DeBBI for Other Libraries

Project	Stars	Client Num	Revision	Bug Num			Execution Time(min)			Test Case		
				Unkown	Known	Discussion	First	Last	Average	First	Last	Average
Commons-io	616	4,308	2.1 - 2.6	1	0	3	5.29 (99.62%)	408.45 (93.50%)	181.6 (95.37%)	245 (99.39%)	3525 (97.49%)	1428 (98.44%)
assertj-core	1,689	1,129	3.8.0 - 3.14.0	2	8	0	0.04 (99.99%)	211.6 (89.01%)	130.86 (88.60%)	1 (99.99%)	5252 (96.01%)	3767 (93.66%)
lombok	8,832	2,721	1.16.14 - 1.18.10	1	10	0	0.34 (99.31%)	227.74 (92.20%)	72.21 (95.39%)	1 (99.79%)	421 (99.25%)	175 (99.41%)
commons-vfs	103	39	2.2 - 2.6.0	1	0	0	0.96 (98.21%)	0.96 (98.21%)	0.96 (98.21%)	112 (92.21%)	112 (92.21%)	112 (92.21%)
jsoup	7,650	575	1.9.2 - 1.12.1	1	2	0	1.64 (99.60%)	39.49 (93.96%)	17.8 (96.56%)	81 (98.18%)	399 (96.25%)	243 (96.46%)
mybatis-spring	1,992	987	1.3.2 - 2.0.3	1	2	0	1.67 (97.30%)	48.29 (91.64%)	32.68 (90.23%)	1 (99.12%)	9 (99.92%)	6 (99.92%)
HttpAsyncClient	904	125	4.1.3 - 4.1.4	1	1	0	4.14 (97.27%)	7.9 (94.91%)	6.02 (96.08%)	55 (98.66%)	78 (98.14%)	66 (98.41%)
JENA	618	59	3.12.0 - 3.14.0	1	0	0	1.27 (98.52%)	1.27 (98.52%)	1.27 (98.52%)	13 (98.57%)	13 (98.57%)	13 (98.57%)
ognl	111	70	3.1 - 3.2.12	1	2	0	0.8 (97.28%)	3.52 (95.66%)	2.34 (95.81%)	5 (99.03%)	37 (98.20%)	19 (98.26%)
asciidoctorj	445	27	1.5.3 - 2.2.0	2	0	0	3.92 (92.09%)	4.4 (92.86%)	4.16 (92.52%)	6 (98.18%)	6 (98.27%)	6 (98.22%)
mybatis	12,730	1,135	3.1.1 - 3.5.3	1	7	0	1.18 (97.85%)	54.67 (93.57%)	11.88 (96.30%)	13 (92.61%)	106 (99.26%)	68 (98.71%)
java-jwt	3,323	119	3.2.0 - 3.6.0	1	1	0	0.63 (85.80%)	4.47 (96.87%)	2.55 (96.53%)	7 (77.42%)	21 (98.47%)	14 (98.00%)
mybatis-generator	4,105	202	1.3.5 - 1.4.0	1	1	1	0.27 (95.15%)	0.66 (98.95%)	0.51 (97.95%)	3 (62.50%)	3 (97.35%)	3 (93.18%)
jOOQ	3,646	88	3.9.0 - 3.12.4	1	1	1	0.74 (97.49%)	6.14 (97.22%)	3.75 (97.11%)	3 (99.23%)	27 (98.77%)	16 (98.75%)
bcpkix-jdk15on	1,110	122	1.5.9 - 1.6.4	1	0	0	12.3 (94.66%)	12.3 (94.66%)	12.3 (94.66%)	169 (97.67%)	169 (97.67%)	169 (97.67%)
activiti-engine	6,239	39	6.0.0 - 7.1.0	1	0	0	0.67 (96.20%)	0.67 (96.20%)	0.67 (96.20%)	1 (98.15%)	1 (98.15%)	1 (98.15%)
extentreports	517	43	3.0.7 - 4.1.2	1	0	0	3.22 (70.51%)	3.22 (70.51%)	3.22 (70.51%)	36 (86.86%)	36 (86.86%)	36 (86.86%)

```
java util Calendar java lang String java util Date java lang Integer java util TimeZone java text
SimpleDateFormat java util Locale java util Map java util ResourceBundle java util Collection java util Set
java lang StringBuilder java util ListIterator java util Iterator java util List java lang Double java lang Class
```

Figure 10: Example changed JDK query

```
~/ViterbiAlgorithm.class:[java/util/Map,java/util/Collection,java/lang/Object,java/lang/StringBuilder,java
/util/Set,java/util/ListIterator,java/lang/String,java/util/Iterator,java/util/List,java/lang/Double]
~/ViterbiAlgorithmTest.class:[java/lang/String,java/util/Collection,java/lang/Object,java/util/Set,java/util
/Iterator,java/util/Map,java/util/List,java/lang/Double]
~/Utils.class:[java/util/Set,java/lang/Object,java/util/Iterator,java/util/Map,java/lang/Double]
~/ForwardBackwardAlgorithmTest.class:[java/lang/String,java/util/Collection,java/lang/Object,java/util/
Map,java/lang/Double,java/util/List]
~/ForwardBackwardAlgorithm.class:[java/util/Collection,java/lang/Object,java/util/ListIterator,java/util/
Set,java/lang/String,java/util/Iterator,java/util/Map,java/util/List,java/lang/Double]
~/SequenceState.class:[java/lang/Object,java/lang/Double]
~/Transition.class:[java/lang/Object,java/lang/StringBuilder,java/lang/String,java/lang/Class]
```

Figure 11: Project hmm-lib JDK usage

```
~ /UmmalquraFormatData_ar.class:[java/lang/Object]
~/UmmalquraGregorianConverterTests.class:[java/util/Calendar, java/lang/String, java/util/Date]
~/UmmalquraDateFormatTests.class:[java/util/Calendar, java/lang/Integer, java/lang/Object,
java/lang/String, java/util/TimeZone, java/text/SimpleDateFormat, java/util/Locale, java/util/Date]
~/UmmalquraCalendar.class:[java/lang/Integer, java/lang/Object, java/lang/String, java/util/TimeZone,
java/util/Map, java/util/Locale, java/util/Date]
~/UmmalquraDateFormatSymbols.class:[java/util/ResourceBundle, java/lang/Object, java/lang/String,
java/util/Locale]
~/UmmalquraFormatData_en.class:[java/lang/Object]
```

Figure 12: Project ummalqura-calendar JDK usage

4 DISCUSSIONS

Availability of Client Software In our experiment, due to the prevalent usage of JDK, we were able to collect 2,953 client software projects, and ran unit testing on them over JDK 8 and 9 to detect failures. One doubt on the applicability of our approach is whether there are also many client software projects for other libraries so that prioritization is necessary. Our observation is that the popular frameworks that require extensive incompatibility detection typically have lots of client software project available. For example, Android SDK, Apache software, Eclipse API, and Chrome API all have thousands of client projects in GitHub (as confirmed in RQ5). On the other hand, due to the popularity of modern build systems (Gradle/Maven) and the corresponding central repositories, even ordinary projects can have a large number of client projects on the central repositories. Such modern build systems support fully automated client project retrieval, build, and test. Thus, we can easily apply DeBBI in a fully automated way[2].

Effectiveness of Client Software Testing Another issue with client software testing is whether it is helpful when a large regression test suite is already available. From our experiment, we can see that 79 JDK incompatibility bugs can be detected if client software testing is applied before Java 9.0.0 is released. These bugs are confirmed by JDK developers in 9.0.1, and cannot be detected by the large regression test suite of JDK. Another benefit of client software testing is that it always finds real bugs. Although regression testing may also detect incompatibilities, the ones detected may be on a cold spot of API that is never used by real client software, or triggered by a method-invocation sequence that is never used by client software developers. In contrast, the incompatibilities detected by client software testing usually indicate important bugs of the library or the client software.

Why does DeBBI work? A naive approach for ranking client projects would be simply counting the number of API terms used by each client project. In contrast to simply counting API term frequency, our DeBBI adopts information retrieval, which not only counts API term frequency, but also considers API importance, diversity, and textual information. For example, there are two JDK client projects hmm-lib [3] and ummalqura-calendar [7] from our data set. Figure 10 shows the portion of changed JDK query which is related to these two client projects, while Figures 11 and 12 show the JDK usage of the client projects. Interestingly, we can see many terms (highlighted in bold) matching terms in query. If we only count the term frequency, hmm-lib with 125 term matches should have a higher priority than ummalqura-calendar that only has 67 term matches. However, in our DeBBI(TF.IDF), hmm-lib is ranked at 2,760 with no bug and ummalqura-calendar is ranked at 442 with a real BBI issue (BugID: JDK-8008577[3], triggered by the different English locale date-time long formats between JDK 8 and JDK 9). The reason is that TF.IDF also considers the importance of low-frequency terms Locale and Date.

Why do we need diversity enhancement? For a given query, an information retrieval system can give us a ranked list of documents all of which are relevant to the query. However, they might be all the same or very similar. This is a classic diversity or novelty issue in information retrieval. In our scenario, if DeBBI uses only traditional information retrieval technique, the top-ranked client projects might detect the same bugs repeatedly. Therefore, we use the MMR algorithm to solve this issue to detect more unique bugs faster. In Figure 13, the solid and dashed lines present the effectiveness of detecting unique BBI bugs for JDK when applying LDA and LDA+MMR, respectively. The x-axis is the number of projects we

[2]We can also afford discarding failing client projects as online repositories provide a huge candidate project set.

[3]https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8008577

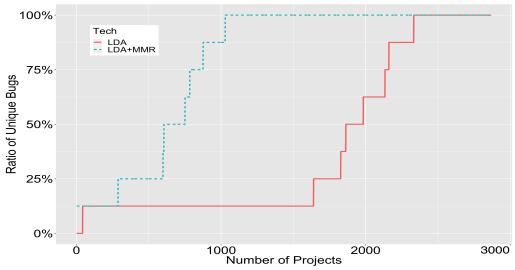

Figure 13: Accumulated bugs detected

need to run, the *y*-axis is the percentage of unique JDK BBI bugs we can detect. We observed that LDA+MMR found the first unique bug at the 1st position and the last unique bug at the 1029rd position, while LDA found the first/last unique bug at the 43rd/2333rd position, demonstrating the effectiveness of diversity enhancement for further boosting DeBBI.

5 RELATED WORKS

5.1 Test Prioritization

Test-case prioritization is a well studied research area. As for generic prioritization strategies, the total and additional strategies are the most widely-used prioritization strategies [70], and reported empirical results show that the additional strategy is more effective than the total strategy in most cases. There also have been a number of research efforts seeking for other optimal prioritization strategies. For example, Li et al. [52] proposed a 2-optimal strategy based on two different strategies: hill-climbing, and genetic programming. respectively. Jiang et al. [42] proposed an adaptive random strategy for test-case prioritization. Bryce and Memon [29] proposed to prioritize test cases (i.e., event sequences) for event-based GUI software. As each test case is an event sequence in GUI testing, their approach tries to select event sequences to cover different event interactions as early as possible. Zhang et al. [85] proposed a generic strategy that has flavor of both total and additional strategies.

Besides proposing generic prioritization strategies, researchers have also investigated test prioritization using different levels of code coverage. There have been research work based on statement and branch coverage [70], function coverage [35], block coverage [34], modified condition/decision coverage [34], etc. There have also been research [44] on test-case prioritization using coverage of system models. Mei et al. [57] investigated criteria based on dataflow coverage for testing service-oriented software. More recently, Saha et al. [72] utilized the textual similarity between tests and code changes based on IR to perform test prioritization. In this paper, we are prioritizing client software projects instead of test cases, and thus we face two very different challenges. First, since it takes huge amount of time to execute tests of all client software projects, our approach must be static (i.e., not using any runtime information). Second, compared with test cases which are designed to cover different parts of a software project, client software projects contain much more redundancy. Therefore, to overcome these challenges, we developed an IR-based approach and further optimized it considering the diversity of term coverage (based on MMR) and test relevance (via extending static RTS).

5.2 Automated Test Generation

Another area that is related to our work is test generation based on existing client code. Suresh et al. [78] proposed an approach to automatically generate test cases by mining source code from client software projects, and later extended the technique with mining of dynamic execution traces [77, 79]. Bozkurt and Harman [28] proposed an approach to generate test cases from web service transactions. Pradel and Gross [63] combined specification mining from client code and test generation to detect API usage bugs. More recently, Ma et. al. [54] proposed to use library test cases to guide test-case generation for client software. Reiss [66] proposed to use semantic code search to find potential client code for test generation. Research efforts in this area focuses on generating test cases for one software project based on source code or test code of the current or other software projects. Therefore, they actually solve a different problem, and suffer from the general problems of test generation, such as the test oracle and unrealistic test (i.e., exploration of method invocation sequences that never happens in reality) problems, when directly used for library code testing. In contrast, our prioritization technique opens a new dimension via utilizing the large number of existing client project tests in the wild for detecting library BBIs, and can be complementary to these existing test generation techniques.

6 CONCLUSION AND FUTURE WORK

In this work, we propose to detect library backward incompatibilities using the large number of client project test suites in the wild, i.e., *cross-project* library upgrade testing. However, it typically involves huge testing efforts. Therefore, we further present a novel approach to prioritizing software projects in large-scale client software testing based on information retrieval, DeBBI. Furthermore, we also further optimize DeBBI via considering the API-use diversity (based on MMR) and test relevance (via extending static RTS). Our evaluation shows that, compared with the baseline random project prioritization, our approach can reduce the time to detect the first and average unique BBI bug by 99.1% and 70.8% for JDK, and detect various previously unknown BBI bugs for popular 3rd-party libraries.

In the future, we plan to further extend our research project from the following directions. First, we plan to carry out our experiments on more software libraries and confirmed bugs to further reduce the threats discussed in Section 3.6. Second, although our evaluation shows that prioritizing projects may largely reduce the cost of large-scale client testing, prioritizing individual test cases in the selected projects may further reduce the cost. Third, we show that client software testing can detect library incompatibilities. However, since client software can be much more complicated than the test code of library, how to report, localize, and fix such failures also becomes challenging, and we plan to further work on these challenging problems.

7 ACKNOWLEDGEMENTS

This work is partially supported by National Science Foundation under Grant Nos. CCF-1763906, CCF-1846467, and CCF-1942430.

REFERENCES

[1] 2007. Criticism of Windows Vista. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en. (2007). Accessed: 2014-08-30.

[2] 2014. Sougou. https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en. (2014). Accessed: 2014-08-30.

[3] 2016. This library implements Hidden Markov Models (HMM) for time-inhomogeneous Markov processes. (2016). https://github.com/bmwcarit/hmm-lib.

[4] 2018. Android Software Development Kit. (2018). https://developer.android.com/.

[5] 2018. Apache Struts. (2018). https://struts.apache.org/.

[6] 2018. ASM Bytecode Manipulation Framework. (2018). http://asm.ow2.org/.

[7] 2018. Implementation of java.util.Calendar for the Umm Al-Qura calendar system. (2018). https://github.com/msarhan/ummalqura-calendar.

[8] 2018. Indri. (2018). http://www.lemurproject.org/indri.php.

[9] 2018. Java Development Kit. (2018). http://www.oracle.com/technetwork/java/javase/downloads/index.html/.

[10] 2018. jDeps. (2018). https://docs.oracle.com/javase/9/tools/jdeps.htm.

[11] 2018. jdom. (2018). http://www.jdom.org/.

[12] 2018. Log4j. (2018). http://logging.apache.org/log4j.

[13] 2018. Square Libraries. (2018). https://github.com/square.

[14] 2018. The Apache Software Foundation. (2018). http://www.apache.org/.

[15] 2019. assertj-core. https://github.com/joel-costigliola/assertj-core/issues/1751. (2019).

[16] 2019. COLLECTIONS-721. https://issues.apache.org/jira/browse/COLLECTIONS-721. (2019).

[17] 2019. httpasyncclient. https://issues.apache.org/jira/browse/HTTPASYNC-159. (2019).

[18] 2019. java-jwt. https://github.com/auth0/java-jwt/issues/376. (2019).

[19] 2019. jena-arq. https://issues.apache.org/jira/browse/JENA-1819. (2019).

[20] 2019. Jsoup. https://github.com/jhy/jsoup/issues/1274. (2019).

[21] 2019. lombok. https://github.com/rzwitserloot/lombok/issues/2320. (2019).

[22] 2019. mybatis-spring. https://github.com/mybatis/spring/issues/427. (2019).

[23] 2019. reflection. https://github.com/ronmamo/reflections/issues/277. (2019).

[24] 2019. vfs. https://issues.apache.org/jira/browse/VFS-7392320. (2019).

[25] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. 2010. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering-Volume 1*. ACM, 95–104.

[26] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 265–279.

[27] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[28] M. Bozkurt and M. Harman. 2011. Automatically generating realistic test input from web services. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. 13–24. https://doi.org/10.1109/SOSE.2011.6139088

[29] Renée C. Bryce and Atif M. Memon. 2007. Test suite prioritization by interaction coverage. In *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting (DOSTA '07)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1294921.1294922

[30] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 335–336.

[31] Kingsum Chow and D. Notkin. 1996. Semi-automatic update of applications in response to library changes. In *Software Maintenance 1996, Proceedings., International Conference on*. 359–368.

[32] Bradley Cossette and Robert J. Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. 55.

[33] D. Dig and R. Johnson. 2005. The role of refactorings in API evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. 389–398.

[34] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. 2004. Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. In *ISSRE*. 113–124.

[35] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00)*. ACM, New York, NY, USA, 102–112. https://doi.org/10.1145/347324.348910

[36] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.

[37] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 211–222.

[38] Jade Goldstein, Vibhu Mittal, Jaime Carbonell, and Mark Kantrowitz. 2000. Multi-document summarization by sentence extraction. In *Proceedings of the 2000 NAACL-ANLP Workshop on Automatic summarization*. Association for Computational Linguistics, 40–48.

[39] Shengbo Guo and Scott Sanner. 2010. Probabilistic latent maximal marginal relevance. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM, 833–834.

[40] Djoerd Hiemstra. 2001. Using language models for information retrieval. (2001).

[41] Liangjie Hong, Ovidiu Dan, and Brian D Davison. 2011. Predicting popular messages in twitter. In *Proceedings of the 20th international conference companion on World wide web*. ACM, 57–58.

[42] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. 2009. Adaptive Random Test Case Prioritization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 233–244. https://doi.org/10.1109/ASE.2009.77

[43] Seokhwan Kim, Yu Song, Kyungduk Kim, Jeong-Won Cha, and Gary Geunbae Lee. 2006. Mmr-based active machine learning for bio named entity recognition. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. Association for Computational Linguistics, 69–72.

[44] Bogdan Korel, Luay Ho Tahat, and Mark Harman. 2005. Test Prioritization Using System Models. In *ICSM*. 559–568.

[45] Ralf Krestel, Peter Fankhauser, and Wolfgang Nejdl. 2009. Latent dirichlet allocation for tag recommendation. In *Proceedings of the third ACM conference on Recommender systems*. ACM, 61–68.

[46] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class firewall, test order, and regression testing of object-oriented programs. *JOOP* 8, 2 (1995), 51–65.

[47] Thomas K Landauer, Peter W Foltz, and Darrell Laham. 1998. An introduction to latent semantic analysis. *Discourse processes* 25, 2-3 (1998), 259–284.

[48] Changki Lee and Gary Geunbae Lee. 2006. Information gain and divergence-based feature selection for machine learning-based text categorization. *Information processing & management* 42, 1 (2006), 155–165.

[49] Dik L Lee, Huei Chuang, and Kent Seamons. 1997. Document ranking and the vector-space model. *IEEE software* 14, 2 (1997), 67–75.

[50] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 583–594.

[51] Hareton KN Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *ICSM*. 290–301.

[52] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Trans. Software Eng.* 33, 4 (2007), 225–237.

[53] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2007. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, 4 (2007), 13.

[54] Lei Ma, Cheng Zhang, Bing Yu, and Jianjun Zhao. 2016. Retrofitting automatic testing through library tests reusing. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4.

[55] Yoëlle S Maarek, Daniel M Berry, and Gail E Kaiser. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering* 17, 8 (1991), 800–813.

[56] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. 70–79.

[57] Lijun Mei, Zhenyu Zhang, W. K. Chan, and T. H. Tse. 2009. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th international conference on World wide web (WWW '09)*. ACM, New York, NY, USA, 901–910. https://doi.org/10.1145/1526709.1526830

[58] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2015. A Study on Behavioral Backward Incompatibilities of Java Software Libraries. In *Proceedings of the 2017 International Symposium on Software Testing and Analysis*. 215–225.

[59] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*. 241–251.

[60] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Christina Lioma. 2006. Terrier: A high performance and scalable information retrieval platform. In *Proceedings of the OSIR Workshop*. 18–25.

[61] Ian Porteous, David Newman, Alexander Ihler, Arthur Asuncion, Padhraic Smyth, and Max Welling. 2008. Fast collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 569–577.

[62] Martin F Porter. 2001. Snowball: A language for stemming algorithms. (2001).

[63] Michael Pradel and Thomas R. Gross. 2012. Leveraging Test Generation and Specification Mining for Automated Bug Detection Without False Positives. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 288–298. http://dl.acm.org/citation.cfm?id=2337223.2337258

[64] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic Versioning and Impact of Breaking Changes in the Maven Repository. *J. Syst. Softw.* 129, C (July 2017), 140–158. https://doi.org/10.1016/j.jss.2016.04.008

[65] Adrian E Raftery and Steven Lewis. 1991. *How many iterations in the Gibbs sampler?* Technical Report. WASHINGTON UNIV SEATTLE DEPT OF STATISTICS.

[66] Steven P. Reiss. 2014. Towards Creating Test Cases Using Code Search. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. 436–440. https://doi.org/10.1109/ICSME.2014.69

[67] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. ACM, 42–49.

[68] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp* 109 (1995), 109.

[69] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.

[70] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*. 179–188.

[71] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *ASE*. 345–355.

[72] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 268–279. https://doi.org/10.1109/ICSE.2015.47

[73] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[74] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. 2014. A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. ACM, 101–110.

[75] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 237–247.

[76] Stephen W Thomas. 2011. Mining software repositories using topic models. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 1138–1139.

[77] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. 2010. DyGen: Automatic Generation of High-Coverage Tests via Mining Gigabytes of Dynamic Traces. In *Tests and Proofs*, Gordon Fraser and Angelo Gargantini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–93.

[78] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-oriented Unit-test Generation via Mining Source Code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 193–202. https://doi.org/10.1145/1595696.1595725

[79] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing Method Sequences for High-coverage Testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 189–206.

[80] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. 2009. Using latent dirichlet allocation for automatic categorization of software. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 163–166.

[81] Yuan Tian, David Lo, and Chengnian Sun. 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 215–224.

[82] Ellen M Voorhees, Donna K Harman, et al. 2005. *TREC: Experiment and evaluation in information retrieval*. Vol. 1. MIT press Cambridge.

[83] Xiaogang Wang and Eric Grimson. 2008. Spatial latent dirichlet allocation. In *Advances in neural information processing systems*. 1577–1584.

[84] Chengxiang Zhai and John Lafferty. 2004. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS)* 22, 2 (2004), 179–214.

[85] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*. 192–201.

[86] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. 23–32.

[87] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*. 14–24.