Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization

Qianyang Peng
University of Illinois
Urbana, IL, USA
qp3@illinois.edu

August Shi
University of Illinois
Urbana, IL, USA
awshi2@illinois.edu

Lingming Zhang
University of Texas at Dallas
Dallas, TX, USA
lingming.zhang@utdallas.edu

ABSTRACT

Test-case prioritization (TCP) aims to detect regression bugs faster via reordering the tests run. While TCP has been studied for over 20 years, it was almost always evaluated using seeded faults/mutants as opposed to using *real test failures*. In this work, we study the recent change-aware information retrieval (IR) technique for TCP. Prior work has shown it performing better than traditional coverage-based TCP techniques, but it was only evaluated on a small-scale dataset with a cost-unaware metric based on seeded faults/mutants. We extend the prior work by conducting a much larger and more realistic evaluation as well as proposing enhancements that substantially improve the performance. In particular, we evaluate the original technique on a large-scale, real-world software-evolution dataset with real failures using both cost-aware and cost-unaware metrics under various configurations. Also, we design and evaluate hybrid techniques combining the IR features, historical test execution time, and test failure frequencies. Our results show that the change-aware IR technique outperforms state-of-the-art coverage-based techniques in this real-world setting, and our hybrid techniques improve even further upon the original IR technique. Moreover, we show that flaky tests have a substantial impact on evaluating the change-aware TCP techniques based on real test failures.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Test-case prioritization, information retrieval, continuous integration

ACM Reference Format:
Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397383

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '20, July 18–22, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397383

1 INTRODUCTION

Test-case prioritization (TCP) aims to detect regression bugs faster by reordering the tests such that the ones more likely to fail (and therefore detect bugs) are run first [50]. To date, researchers have proposed a large number of TCP techniques, including both *change-unaware* and *change-aware* techniques. Change-unaware techniques use dynamic or static information from the old version to perform TCP. For example, the *total* technique simply sorts all the tests in the descending order of the number of the covered program elements (e.g., methods or statements), while the improved *additional* technique sorts the tests based on the number of their covered elements that are not covered by already prioritized tests [51]. In contrast, change-aware techniques consider program changes between the old and new software versions to prioritize tests more likely to detect bugs due to these changes. For example, a change-aware technique based on information retrieval (IR) [53] reduces the problem of TCP into the traditional IR problem [37]—the program changes between revisions are treated as the *query*, while the tests are treated as the *data objects*; the tests that are textually more related to the program changes are prioritized earlier.

Although various studies have investigated the effectiveness of existing TCP techniques, they suffer from the following limitations. First, they are usually performed on artificial software evolution with seeded artificial or real bugs, and not on real-world software evolution with *real test failures*. For example, recently Luo *et al.* [34] empirically evaluated TCP techniques on both artificial bugs via mutation testing and real bugs from Defects4J [22]. However, even Defects4J bugs are not representative of real regression bugs, because Defects4J bugs are isolated bugs, while real regression bugs often come with other benign changes. Furthermore, developers using TCP as part of their workflow would not know a priori what are the exact bugs in the code; they would only be able to observe the test failures that occur. Second, the existing TCP techniques are usually evaluated using *cost-unaware metrics*, such as Average Percentage of Faults Detected (APFD) [50]. Even state-of-the-art TCP techniques were also evaluated using only APFD [33, 44, 53], which can be biased and unrealistic [10, 36]. Last but not least, *flaky tests* have been demonstrated to be prevalent in practice [32], but to the best of our knowledge, none of the existing studies of TCP considered the impacts of flaky tests. In evaluating TCP effectiveness, flaky test failures would not indicate real bugs in the code, so having such failures can mislead the results if the goal is to order the tests that fail (because presumably they detect a real bug) first.

In this work, we focus on the IR technique for TCP based on program changes [53], which we call the *IR-based TCP technique*. This technique uses IR to calculate the textual similarity between the program changes and each test, then prioritizes the tests in

descending order of their similarities with the changes. We focus on this specific technique because it has been shown to outperform traditional change-unaware total and additional techniques [53]. Furthermore, the IR-based TCP technique is a lightweight static technique that scales well and can be easily applied to a large set of projects. In our new evaluation, we aim to reduce the limitations of the prior studies by using a dataset consisting of real test failures in projects that occur due to real software evolution and by using both cost-unaware and cost-aware metrics for evaluation. We also aim to further enhance the IR-based TCP technique by exploring more configurations and hybridizing IR features with historical test execution time and failure frequencies to improve its performance.

Some other studies have started using real test failures for other aspects of regression testing, e.g., for test-suite reduction [56] regression test selection (RTS) [7], and the impact of flaky tests [25]. Elbaum *et al.* proposed their TCP and RTS techniques and evaluated them on a closed-source Google dataset [14]. In contrast, we construct a dataset using real-world, open-source projects.

This paper makes the following contributions:

- *Dataset:* We collect a large-scale dataset from GitHub and Travis CI. Our dataset consists of 2,042 Travis CI builds with 2,980 jobs (a Travis CI build can have multiple jobs), and 6,618 real test failures from 123 open-source Java projects. For each job, our dataset has the source code, program changes, and the test execution information (the pass/fail outcome and the test execution time), which suffice for evaluating TCP techniques. Our dataset also includes the test coverage information for 565 jobs and the test flakiness labeling for 252 jobs. Moreover, our dataset is publicly available [4].
- *Novel Technique:* We propose new hybrid IR-based TCP techniques that takes into consideration not only change-aware IR but also test execution time and historical test failure frequencies. Our hybrid IR-based TCP techniques significantly outperform the traditional IR-based TCP techniques and history-based techniques.
- *Evaluation:* We evaluate a wide range of configurations of IR-based TCP techniques on our dataset using both cost-unaware and cost-aware metrics. We are among the first to evaluate TCP techniques based on a large number of real test failures taken from real-world project changes as opposed to seeded mutants and faults in the code. We are the first to evaluate specifically IR-based TCP techniques with a cost-aware metric, and we are the first to study the impacts of flaky tests on TCP, which pose a problem to evaluations based on test failures.
- *Outcomes:* Our study derives various outcomes, including: (1) there is a huge bias when using a cost-unaware metric for TCP instead of a cost-aware metric; (2) the IR-based TCP technique outperforms state-of-the-art coverage-based TCP techniques on real software evolution by both cost-unaware and cost-aware metrics; (3) our hybrid IR-based TCP techniques are very effective in performing TCP and significantly outperform all traditional IR-based TCP techniques and history-based techniques; and (4) flaky tests have a substantial impact on the change-aware TCP techniques.

2 INFORMATION RETRIEVAL (IR) TECHNIQUES

Saha *et al.* [53] first proposed using information retrieval (IR) techniques to perform test-case prioritization (TCP). The key idea is that tests with a higher textual similarity to the modified code are more likely related to the program changes, thus having a higher chance to reveal any bugs between the versions due to the changes.

In general, in an IR system, a *query* is performed on a set of *data objects*, and the system returns a ranking of the data objects based on similarity against the input query. There are three key components for an IR system: (1) how to construct the data objects, (2) how to construct the query, and (3) the retrieval model that matches the query against the data objects to return a ranking of the data objects as the result. We describe each component in detail and how they relate to performing TCP.

2.1 Construction of Data Objects

For IR-based TCP, the data objects are constructed from the tests[1]. To construct the data objects, the first step is to process the string representation of each test into tokens. A *token* is a sequence of characters that act as a semantic unit for processing. For example, a variable can be a token. We consider four different approaches to process the test files.

The first and most naive approach we consider is to use the whitespace tokenizer that breaks text into tokens separated by any whitespace character. We denote this approach as *Low*. The approach is simple and straightforward, but it results in two main disadvantages: (1) the approach does not filter out meaningless terms for IR such as Java keywords (e.g., *if*, *else*, *return*), operators, numbers and open-source licenses; and (2) the approach fails to detect the similarities between the variable names that are partially but not exactly the same, e.g., $setWeight$ and $getWeight$.

An optimization for the naive *Low* approach is to use a special code tokenizer to tokenize the source code; we denote this second approach as Low_{token}. Low_{token} improves upon *Low* by introducing a code tokenizer that (1) filters out all the numbers and operators, (2) segments the long variables by non-alphabetical characters in-between and by the camel-case heuristics, and (3) turn all upper-case letters to lower-case letters. With this code tokenizer, Low_{token} removes several of the disadvantages of *Low*. However, Low_{token} still does not filter out some meaningless terms for IR, such as Java keywords and open-source licenses.

A third approach is to build an abstract syntax tree (AST) from each test file and extract the identifiers and comments [53]. This step removes most meaningless terms. We denote this approach as *High*. After extracting the identifiers, we can additionally apply the same code tokenizer as Low_{token} to create finer-grained tokens. We denote this fourth, final approach as $High_{token}$.

2.2 Construction of Queries

IR-based TCP constructs a query based on the program changes (in terms of changed lines) between two versions. Similar to the construction of data objects, we also apply one of *Low*, Low_{token}, *High*, and $High_{token}$ as the preprocessing approach for the query.

[1] In this work, we prioritize test classes as tests; a test class can contain multiple test methods. In the rest of this paper, when we refer to a test, we mean test class.

For each type of data-object construction, we apply the same corresponding preprocessing approach, e.g., if using $High_{token}$ for data objects, we use $High_{token}$ for the query as well.

Example 1: The context of changed code

```
1   private void initEnvType() {
2     // Get environment from system property
3     m_env = System.getProperty("env");
4 +   if (Utils.isBlank(m_env)) {
5 +     m_env = System.getProperty("active");
6 +   }
7     if (!Utils.isBlank(m_env)) {
8       m_env = m_env.trim();
9       logger.info(info, m_env);
10      return;
11    }
```

An important consideration for constructing the query is whether to include the *context* of the change or not. By context, we mean the lines of code around the exact changed lines between the two versions. Example 1 illustrates a small code diff with the surrounding context, taken from a change to file `DefaultServerProvider.java` in GitHub project `ctripcorp/apollo` [2]. In the example, lines 4-6 are the modified lines (indicated by +), and the highlighted lines show the considered context for the diff if configured to include 1 line of context. More context makes the query more informative, and the query has a higher chance to reveal hidden similarities between the code change and tests. However, more context also has a higher chance to include unrelated text into the query. Prior work [53] used 0 lines of context to construct queries without evaluating the impact of this choice of context. In our evaluation, we evaluate the impact of including 0 lines, 1 line, 3 lines, 5 lines of context before and after the changed lines, and we also consider including the contents of the whole file as the context.

2.3 Retrieval Models

The retrieval model part of the IR system takes the data objects and the query as input, and it generates a ranking of data objects (i.e., tests for IR-based TCP) as the output. In our evaluation, we explore the following four retrieval models: Tf-idf, BM25, LSI and LDA.

Tf-idf. Tf-idf [54] is a bag-of-words based text vectorization algorithm. Given the vector representations of the data objects, we perform the data-object ranking based on the vector distances between the data objects' vectors and the query. We use TfidfVectorizer from scikit-learn [45] to implement the Tf-idf model. We perform TCP by sorting the tests in a descending order of the cosine similarity scores between their Tf-idf vectors and the query's vector.

BM25. BM25 [48] (also known as Okapi BM25), is another successful retrieval model [47]. Unlike the Tf-idf model, BM25 takes the data-object lengths into consideration, such that shorter data objects are given higher rankings. Moreover, unlike the traditional Tf-idf model, which requires using the cosine similarity scores to perform the ranking after feature extraction, BM25 itself is designed to be a ranking algorithm that can directly compute the similarity scores. In our evaluation, we use the gensim [46] implementation of BM25, and we use their default parameters. We perform TCP by sorting tests in the descending order of their BM25 scores. Saha *et al.* [53] previously evaluated IR-based TCP using BM25 on Java projects, and recently Mattis and Hirschfeld also evaluated IR-based TCP using BM25, but on Python projects.

LSI and LDA. LSI [11] and LDA [62] are two classic unsupervised bag-of-words topic models. As a topic model, the representation of a data object or a query is a vector of topics rather than a vector of raw tokens. This mathematical embedding model transforms the data objects from a very high dimensional vector space with one dimension per word into a vector space with a much lower dimension. Using a vector of topics to represent each data object and the query, we can calculate the similarity scores and rank the data objects. We use the gensim implementation of LSI and LDA, and we use cosine similarity to calculate vector similarities to compute the ranking.

2.3.1 New Tests and Tie-Breaking. In the case of new tests, which means there is no IR score, we configure IR-based TCP to place the new tests at the very beginning of the test-execution order, because we believe new tests are the most related to the recent changes.

When multiple tests share the same score, it means that IR-based TCP considers them equally good. However, the tests still need to be ordered. In such a scenario, we order these tests deterministically in the order they would have been scheduled to run when there is no TCP technique that reorders them (basically running them in the order the test runner runs them).

3 CONSTRUCTING DATASET

To perform IR-based TCP, we need the code changes between two software versions (for the query) and the textual contents of tests (for the data objects). For our evaluation, we need test execution outcomes (pass or fail) and test execution time as to evaluate how well an IR-based TCP technique orders failing tests first. We construct our dataset that contains all of the above requirements.

To construct our dataset, we use projects that use the Maven build system, are publicly hosted on GitHub, and use Travis CI. Maven [2] is a popular build system for Java projects, and the logs from Maven builds are detailed enough to contain the information we need concerning test execution outcomes and times. GitHub [1] is the most popular platform for open-source projects. If a project is active on GitHub, we can download any historical version of the project, and we can get the code changes between any two code versions in the form of an unified diff file. Travis CI [3] is a widely-used continuous-integration service that integrates well with GitHub. When a project on GitHub is integrated with Travis CI, every time a developer pushes or a pull request is made to the GitHub project, that push or pull request triggers a Travis *build*, which checks out the relevant version of the project onto a server in the cloud and then proceeds to compile and test that version.

Each build on Travis CI can consist of multiple *jobs*. Different jobs for the same build run on the same code version, but can run different build commands or run using different environment variables. Each job outputs a different job log and has its own job state (passed, failed, or errored). As such, we treat jobs as the data points for running TCP on in our study, as opposed to builds. We use the Travis CI API to obtain the state of each job as well as the state of the jobs from the previous build. The Travis CI API also

[2] https://github.com/ctripcorp/apollo/pull/1029/files

provides for us the Git commit SHA corresponding to the build and the previous build as well. We use the commit SHAs to in turn query the GitHub API to obtain the diff between the two versions. Furthermore, we use the Travis CI API to download full job logs corresponding to each job. We obtain the test execution outcomes and test execution times for each job by parsing the historical job logs, as long as the job log is well formatted.

We start with 10,000 Java projects collected using GitHub's search API. A single query to the GitHub search API returns at most 1,000 projects, so we use 10 separate sub-queries by the year of creation of the project. In the search query, we specify the primary language to be Java and sort the result by the descending number of stars (a representation of the popularity of a project). Out of the 10,000 projects, we find 1,147 projects that build using Maven and also use Travis CI. For each of these projects, we select the jobs for the project that satisfy the requirements below:

(1) The SHA of the build's corresponding commit is accessible from GitHub, so we can download a snapshot of the project at that commit in the form of a zip file.
(2) The state of the job is "failed" due to test failures, while the state of the previous Travis build is "passed" (A "passed" Travis build means the states of all jobs in it are "passed"). This requirement is to increase the likelihood that the job contains regressions that are due to the code changes.
(3) The logs of both the job and the previous job are analyzable, which means we can get the fully-qualified name (FQN), the outcome (passed or failed), and the execution time of every test run in the jobs.
(4) The code changes between the build and the previous build contains at least one Java file, and the Java files are decodeable and parseable.

For each job log, we extract the FQNs and execution times of executed tests by matching the text with regular expressions "Running: (.*?)" and "Time elapsed: (.*? s)". We extract the failed test FQNs with the provided toolset from TravisTorrent [9].

Given these requirements for jobs, we obtain a final dataset that consists of 2,980 jobs from 2,042 Travis builds across 123 projects. More details on the specific projects and the jobs we analyze can be found in our released dataset [4].

4 EXPERIMENT SETUP

We evaluate IR-based TCP on our collected dataset of projects and test failures. In this section, we discuss the metrics we use to evaluate TCP and give details on the other TCP techniques we compare IR-based TCP against.

4.1 Evaluation Metrics

We use two metrics, Average Percentage of Faults Detected (APFD) and Average Percentage of Fault Detected per Cost (APFDc), to evaluate TCP effectiveness.

4.1.1 Average Percentage of Faults Detected (APFD). Average Percentage of Faults Detected (APFD) [50] is a widely used cost-unaware metric to measure TCP effectiveness:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n}.$$

In the formula, n is the number of tests and m is the number of faults. TF_i is the position of the first failed test that detects the ith fault in the prioritized test suite.

4.1.2 Average Percentage of Fault Detected per Cost (APFDc). Average Percentage of Fault Detected per Cost (APFDc) is a variant of APFD that takes into consideration the different fault severities and the costs of test executions [12, 36]. Due to the difficulty of retrieving the fault severities, prior work typically uses a simplified version of APFDc that only takes into consideration the test execution time [10, 15]. Our evaluation utilizes this simplified version of APFDc in our evaluation:

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

In the formula, n, m, and TF_i have the same definitions as in the formula for APFD, while t_j is the execution time of the jth test.

Prior TCP work [10, 36] has shown that the classic TCP evaluation metric APFD can be very biased and unreal. More specifically, a technique that results in high APFD does not necessarily result in high APFDc, thus being not cost-efficient. Recently, Chen *et al.* [10] also showed that a simple cost-only technique outperforms coverage-based TCP techniques. In this work, we measure TCP effectiveness using both metrics, in particular evaluating IR-based TCP, which was only evaluated using APFD in the past [53]. We measure both as to compare how the two metrics can lead to different results in which TCP technique is the best.

4.1.3 Failure-to-Fault Mapping. Note that APFD and APFDc are defined with respect to the number of *faults* as opposed to failed tests. Prior work evaluated using seeded faults and mutants (treated as faults), with an exact mapping from test failures to faults in the code. However, in our evaluation, we only have information concerning which tests failed, and we do not have an exact mapping from failures to faults. Potentially, a single failed test can map to multiple faults in the code, and conversely multiple failed tests can all map to a single fault. In general, a developer would not know this exact mapping without deep investigation, and given our use of a dataset using real test failures, we also do not know the exact mapping. When answering our research questions, we assume that each single failed test maps to a single distinct fault, similar to one of the mappings Shi *et al.* used in their prior work on test-suite reduction [56], so the number of faults is then the number of test failures. In Section 5.5, we evaluate the impacts of a new mapping that all failures map to the same fault (only one fault in the code), the other extreme for mapping failures to faults.

4.2 Other TCP Techniques

We implement two simple TCP techniques, based on test execution time or historical failures, as baselines to compare IR-based TCP against. While these baselines are relatively simple, prior work has also shown them to be quite effective TCP techniques [10, 14]. We also implement four traditional coverage-based TCP techniques for comparison purposes as well. In general, all these other TCP techniques prioritize based on some metric. For ties in the metric, like with our implementation IR-based TCP, we order the tests deterministically in the order they would have been scheduled to run without any TCP technique in use.

4.2.1 QTF. A simple way to prioritize tests is based on the execution time of the tests. The intuition is that if the fastest-running tests can already find bugs, then running them first should lead to quicker detection of bugs. *QTF* prioritizes the tests by the ascending order of their execution time in the previous job, running the quickest test first. Chen *et al.* found this technique to be a strong baseline in their prior work [10].

4.2.2 HIS. Both industry [35, 40] and academia [7, 42, 59] have considered historical test failures when optimizing software testing techniques. In fact, Maven Surefire, the default unit testing plugin for Maven builds, has a built-in option for specifying that recently failed tests be run first [5]. The intuition is that more frequently failed tests are more likely to fail again in the future. *HIS* builds upon that intuition and prioritizes the tests based on the number of times the test has failed in prior jobs.

To implement HIS, we collect the historical failure frequency data for each test in each job by counting the number of times that the specific test has ever failed before the specific job. Each test t_i is then assigned a historical failure frequency hf_i, indicating the number of times it failed before, and HIS orders the tests by the descending order of hf_i.

4.2.3 Coverage-Based TCP. Coverage-based TCP techniques have been extensively evaluated in empirical studies [31, 50] and are widely used as the state-of-the-art techniques to compare new techniques against [44, 53]. We focus on four traditional coverage-based TCP techniques: (1) Total Test Prioritization [50], (2) Additional Test Prioritization [50], (3) Search-Based Test Prioritization [30, 31], and (4) Adaptive Random Test Prioritization (ARP) [21]. These four techniques are all change-unaware, cost-unaware TCP techniques that aim to find an ideal ordering of the tests based on the coverage information of code elements (statements/branches/methods) [31] obtained on the initial version. We utilize Lu *et al.*'s implementation of the four coverage-based TCP techniques [31].

For every build in our dataset, we collect coverage on the corresponding previous build. Prior work has found that the distance between the version for coverage collection and the version for test prioritization can greatly impact test prioritization results, as the tests and coverages would change over software evolution [13, 31]. In our evaluation, we compute the coverage at the build right before the build where the prioritized test ordering is evaluated, thus maximizing the effectiveness of the coverage-based techniques. We evaluate coverage-based TCP at its best for comparison purposes against IR-based TCP.

To collect coverage on each previous build, we download the corresponding previous build and try to rebuild it locally. We use a clean Azure virtual machine installed with Ubuntu 16.04, and we utilize the default Maven configuration to run the tests. We use OpenClover [43], a coverage collection tool, to collect the coverage information. OpenClover collects coverage per each test method, and we merge the coverage of each test method in the same test class together to form the coverage for each test class. Note that our local build might be different from the original build on Travis CI because of different Maven configurations, and the differences can make the local Maven build crash or produce no result, so we are unable to rerun all the jobs in our dataset locally. We also have to filter out the tests that are executed locally but not recorded in the original build log, to make the test suite consistent between our local build and the original build run on Travis CI. (Such problems with building also illustrate difficulties in implementing and deploying an actual coverage-based TCP technique to run during regression testing.) We successfully rebuild 565 jobs across 49 projects, out of the 2,980 jobs in the dataset. When we compare these coverage-based TCP techniques against IR-based TCP, we compare them only on this subset of jobs.

4.3 Flaky Tests

Our evaluation of TCP using real test failures from historical test executions runs the risk of *flaky tests*. Flaky tests are tests that can non-deterministically pass or fail even for the same code under test [32]. The effect of flaky tests can be substantial, especially for change-aware TCP techniques. If a test failure is due to flakiness, the failure may have nothing to do with the recent change, and a change-aware TCP technique may (rightfully) rank such a test lower. However, if measuring TCP effectiveness based on test failures, a TCP technique can be evaluated as worse than what it should be.

To study the effect of flaky tests on TCP evaluation, we build a dataset that splits test failures into two groups: those definitely due to flaky tests and those likely due to real regressions introduced by program changes. Identifying whether a test failure is due to a flaky test is challenging and would in the limit require extensive manual effort [8, 26]. We use an automated approach by re-running the commits for the jobs with failed tests multiple times and checking what tests fail in those reruns. We select 252 jobs that contain exactly one single test failure and with a build time of less than five minutes, which we rerun six times on Travis CI using exactly the same configuration as when the job was originally run. (We choose only single test failures as to simplify our analysis of categorizing a failure as due to flakiness and a short build time as to not abuse utilizing Travis CI resources.) After the reruns, we record whether the single failure re-occurred or not. A test that does not consistently fail is definitely flaky, while one that does fail in all reruns is likely to be a failing test that indicates a real bug. (Note that having a consistent failure in six reruns does *not* necessarily mean the test is non-flaky.) For a job that we rerun, if the only failure in it is flaky, that job is a *flaky job*; otherwise, it is a *non-flaky job*. As a result, we collect 29 flaky jobs and 223 non-flaky jobs, and then we compare the effectiveness of TCP techniques on them.

5 EMPIRICAL EVALUATION

Our evaluation aims to answer the following research questions:

- **RQ1:** How do different information retrieval configurations impact IR-based TCP techniques in real software evolution?
- **RQ2:** How do IR-based TCP techniques compare against other TCP techniques, in terms of both cost-unaware and cost-aware metrics?
- **RQ3:** How can we further enhance IR-based TCP techniques?
- **RQ4:** How do flaky tests impact the evaluation of TCP in real software evolution?

5.1 RQ1: IR Configurations

The goal of this RQ is to explore different configurations for IR-based TCP; we then use the configuration that has the best performance as the default implementation of the IR-based TCP technique for the later RQs. When comparing the different configurations, we use APFD as the evaluation metric. IR-based TCP is fundamentally cost-unaware, so measuring APFD would better reflect the differences in quality due to differences in the configurations.

As described in Section 2, there are three main configuration options we need to set when implementing an IR-based TCP technique: (1) the code preprocessing approach for data objects and queries (by default using $High_{token}$), (2) the amount of context to the program changes for constructing the query (by default using the whole file as the context), and (3) the retrieval model (by default using BM25, used in prior work [53]). We evaluate which values to set for each option in RQ1.1, 1.2 and 1.3, respectively.

5.1.1 RQ1.1: Code Preprocessing. We apply the four types of code preprocessing, *Low*, Low_{token}, *High*, $High_{token}$, to the tests and program changes for each failed job collected in our dataset. While we change the code preprocessing approach, we set the other parts to their default values. In other words, we use the whole-file strategy for the amount of context to the program changes and use BM25 as the retrieval model.

Table 1: Comparing different preprocessing approaches

Method	APFD			Time on IR (s)		Avg. v-length	
	mean	median	group	mean	median	data	query
Low	0.696	0.757	C	1.294	0.127	10419	3309
Low_{token}	0.755	0.849	A	2.194	0.326	1753	928
High	0.737	0.831	B	0.198	0.044	3056	677
$High_{token}$	0.752	0.844	AB	0.558	0.160	1011	358

Table 1 shows the effectiveness of IR-based TCP when changing the preprocessing approach measured in APFD (we show mean and median), the time spent on running the IR phase, and the mean vocabulary size, which is the number of distinct tokens in the data objects and in the query. Also, we perform a Tukey HSD test [61] for the APFD values, to see if there is any statistical difference between the approaches; the results are shown under the "group" column. The capital letters A-D are the results of the Tukey HSD test, where techniques are clustered into the letter groups, and A is the best while D is the worst (and having two letters means the technique is in a group that is between the two letter groups). We highlight the best approach in gray for each metric, so the highlighted boxes have either the highest mean or median APFD, the shortest mean or median time for running the IR phase, or the shortest vocabulary length for the data objects or queries.

From the table, we make the following observations. First, code tokenization is effective at reducing the vocabulary length. For example, comparing Low_{token} against *Low*, code tokenization reduces the vocabulary size of the data objects by 83.2%; comparing $High_{token}$ against *High*, it reduces the vocabulary size of the data objects concerning extracted identifiers by 66.9%. Second, the approaches utilizing code tokenization usually take longer to run because the time complexity of the BM25 technique is linear to the number of words in the query. That is, although code tokenization reduces the vocabulary size, it makes the data objects and queries longer by splitting long variable names into shorter tokens.[3] Finally, Low_{token}, *High*, and $High_{token}$ lead to similar APFD values, while *Low* has clearly worse results than the other three approaches. The Tukey HSD test shows that Low_{token} performs the best, with it being in its own group and at the top (A). However, $High_{token}$ performs almost as good (and ends up in the second-highest group, AB). $High_{token}$ also runs faster and leads to smaller vocabulary lengths than Low_{token}. Overall, the $High_{token}$ approach is slightly worse than Low_{token} for APFD, but it seems to provide the best trade-offs between APFD, time, and space among all four approaches.

Table 2: Comparing different change contexts

Context	APFD			# Ties	
	mean	median	group	mean	median
0 line	0.697	0.788	C	35.3	2
1 line	0.703	0.788	BC	30.9	2
3 lines	0.713	0.799	BC	25.8	1
5 lines	0.717	0.802	B	23.4	1
Whole-file	0.752	0.844	A	5.3	0

5.1.2 RQ1.2: Context of Change. Table 2 shows the evaluation result of how different lines of context influences the APFD and the number of ties. (Like with evaluating code preprocessing configuration, we set the other configurations to the default values, i.e., code preprocessing is set to $High_{token}$ and the retrieval model is BM25.) Using the whole file as the context has the dominantly best performance in terms of both the highest APFD and the lowest rate of ties, whereas using 0 lines of context leads to the worst performance. We note that the mean value for number of ties is large relative to the median number of ties because of the few jobs that have an extremely large number of ties, which drags the mean up. In general, we observe that IR-based TCP performs better when there are more lines of context in the query.

One interesting observation is that when the length of the query is small, the IR scores for data objects tend to be identical (usually 0), so the prioritization barely changes the ordering of the tests from the original order. Similarity ties are common when we include 0 lines of context into the query. Ideally, the amount of context to include should lead to high APFD values but with a low rate of ties.

5.1.3 RQ1.3: Retrieval Model. We compare the four retrieval models described in Section 2.3. To choose an appropriate retrieval model, we perform a general evaluation of the overall mean and median APFD value across all jobs using each retrieval model. We also perform a per-project evaluation to see for each retrieval model the number of projects that perform the best (having the highest mean APFD across their jobs) with that model.

Table 3 shows the comparison between the different retrieval models. In this table and in the following tables, *#B.Projs* shows the number of projects that have the highest averaged metric value for each specific technique. We can see that BM25 model has both the highest mean and median APFD, and over 60% of the projects in our evaluation have the best results when using BM25.

[3]Note that we do not remove duplicated tokens, because prior work has found doing so would lead to worse results [53].

Table 3: Comparing different retrieval models

Retrieval Model	APFD			
	mean	median	group	#B.Projs
Tf-idf	0.728	0.812	B	27
BM25	0.752	0.844	A	75
LSI	0.707	0.775	C	13
LDA	0.659	0.735	D	8

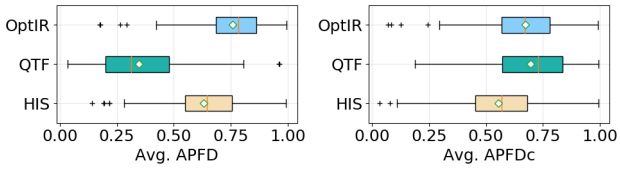
These results shows that BM25 performs better than Tf-idf, indicating that when two tests are equally similar to the same code change, running the test with the fewer lines of code early detects failures faster. Another important observation is that the vector space models (Tf-idf and BM25) perform better than the topic models (LSI and LDA) in terms of APFD, which is consistent with the conclusions found in previous work [52, 67]. The reason for this difference between types of models is likely because the topic models need many more tests to train on. Another likely reason is that the individual tests are not distinct enough between each other, making it hard for the models to generate effective topics.

> ***RQ1:*** *Overall, we determine that the best configuration options to set of IR-based TCP is to use $High_{token}$ as the code preprocessing approach, to use the whole file as the context to include in the queries, and to use BM25 as the retrieval model.*

For the following RQs, we evaluate IR-based TCP using the recommended configurations. We denote IR-based TCP under these optimized configuration values as **OptIR**.

5.2 RQ2: Comparison of IR-Based TCP against Other TCP Techniques

For this RQ, we compare the effectiveness of OptIR against the other TCP techniques (Section 4.2). We measure both APFD and APFDc for evaluating the effectiveness of TCP.



TCP Technique	Avg. APFD			
	mean	median	group	#B.Projs
OptIR	0.759	0.784	A	86
QTF	0.348	0.314	C	7
HIS	0.632	0.645	B	30

TCP Technique	Avg. APFDc			
	mean	median	group	#B.Projs
OptIR	0.672	0.671	A	49
QTF	0.696	0.729	A	53
HIS	0.556	0.568	B	21

Figure 1: Comparing OptIR, QTF, and HIS APFD/APFDc

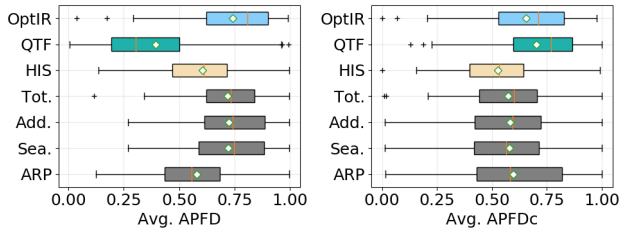
5.2.1 Comparison with QTF and HIS. Figure 1 shows the results of the comparison between OptIR with QTF and HIS in terms of APFD and APFDc. The box plots show the distribution of average APFD and APFDc values per project for each TCP technique. The table in Figure 1 shows the mean and median for the average APFD and APFDc values across projects for each technique. The table also includes the number of projects where that technique results in the best average APFD or APFDc value. Furthermore, we perform a Tukey HSD test to check for statistically significant differences between the three techniques.

From the figure, we observe that the cost-unaware metric APFD can severely over-estimate the performance of cost-unaware techniques and under-estimate the performance of cost-aware techniques. If we take the mean APFDc value as the baseline, the mean APFD over-estimates the performance of OptIR by 12.9% and underestimates the performance of QTF by 50.0%. Also, although QTF performs the worst when evaluated by APFD, it becomes better than both OptIR and HIS when evaluated by APFDc. The mean and median APFDc values for QTF are slightly higher than for OptIR, although the Tukey HSD test suggests that they belong in the same group (A). We further perform a Wilcoxon pairwise signed-rank test [63] to compare the pairs of APFDc values per project between OptIR and QTF, and we find the p-value is 0.339, which does not suggest statistically significant differences between the two. We cannot say then that OptIR is superior to QTF (especially since QTF has higher mean and median APFDc values). Concerning HIS, we clearly see it does not perform well in terms of APFDc. The main reason is because 45.1% of the failures in our dataset are first-time failures for a test, so there is often not any history for HIS to use.

5.2.2 Comparison with Coverage-Based TCP Techniques. In Figure 2, we further compare against traditional coverage-based TCP techniques (Section 4.2.3). The figure shows box plots and a table similar to in Figure 1, except it also includes coverage-based TCP techniques. Recall that we could only collect the test coverage information on a subset of the full dataset (565 jobs across 49 projects), so the results for OptIR, QTF, and HIS shown are different from what are shown in the previous figure. Besides the mean and median averaged APFD/APFDc by each project, we further perform a Tukey HSD test to check for statistically significant differences between all seven TCP techniques. Note that the sum of all the values for #B.Projs does not necessarily add up to the total 49 projects for this table. The reason is that when there is a tie between techniques for the best APFD/APFDc value for a project, we count that project for all the techniques that tie.

From the results, we observe that OptIR, even in this smaller dataset, still performs better than QTF and HIS based on APFD. OptIR also performs better than coverage-based techniques by APFD as well, based on the higher mean and median values. However, while the APFD value is higher, the Tukey HSD test shows that OptIR is in the same (A) group as all the coverage-based TCP techniques, except for ARP (which performs worse). Furthermore, like before, QTF performs the worst based on APFD.

However, when we consider APFDc, once again, QTF outperforms the other TCP techniques, including the coverage-based ones. QTF has the highest mean and median APFDc values, and the Tukey HSD test shows QTF in its own group (A) that is above the groups



TCP Technique	Avg. APFD			
	mean	median	group	#B.Projs
OptIR	0.740	0.807	A	21
QTF	0.393	0.305	C	6
HIS	0.606	0.610	AB	8
Tot.	0.719	0.734	A	12
Add.	0.725	0.740	A	9
Sea.	0.722	0.746	A	8
ARP	0.580	0.555	B	6

TCP Technique	Avg. APFDc			
	mean	median	group	#B.Projs
OptIR	0.656	0.711	AB	14
QTF	0.701	0.768	A	19
HIS	0.528	0.532	B	5
Tot.	0.573	0.601	AB	6
Add.	0.582	0.595	AB	4
Sea.	0.578	0.564	AB	6
ARP	0.595	0.584	AB	4

Figure 2: Comparing all TCP techniques' APFD/APFDc

designated for all the other techniques. When we do a series of Wilcoxon pairwise signed-rank tests between QTF APFDc values and the values for all the other techniques, we see that the differences are statistically significant for all except for OptIR, once again. Also, OptIR ends up in the same Tukey HSD group as all the coverage-based TCP techniques.

Another observation we make is that OptIR in general suffers from a smaller performance loss when evaluation is changed from using APFD to APFDc, compared to coverage-based TCP techniques. The reason is that high-coverage tests tend to be more time-consuming, while higher IR-similarity tests related to the changes do not have such a tendency. This tendency makes traditional coverage-based TCP techniques have relatively lower APFDc values, as more costly tests are prioritized earlier. The ARP technique prioritizes tests based on the diversity of the test coverages rather than the amount of test coverages, so the difference from switching to APFDc is not as prevalent for this technique as well.

> ***RQ2:*** *QTF outperforms almost all other techniques, including traditional coverage-based TCP techniques, suggesting that future work on improving TCP should be compared against and at least perform better than QTF. OptIR performs similarly as QTF when measured by APFDc. OptIR also does not suffer much of a difference in performance when switching the evaluation metric from APFD to APFDc.*

5.3 RQ3: Hybrid Techniques

From RQ2, we find that OptIR does not outperform the simple QTF technique based on APFDc, so we aim to improve OptIR by making it cost-aware. Prior work has tried to address the cost-inefficiency of coverage-based TCP techniques by implementing Cost-Cognizant coverage-based techniques that balance the coverage and the test-execution time, e.g., state-of-the-art Cost-Cognizant Additional Test Prioritization prioritizes tests based on the additional coverage per unit time [10, 15, 36]. However, recent work found that these techniques perform similarly or even worse than the cost-only technique, i.e., QTF. The possible reason is that traditional coverage-based TCP techniques tend to put more costly tests (those with more coverage) earlier, which contrasts with the goals of the Cost-Cognizant feature that tends to put less costly tests earlier. The change-aware IR techniques we study are based on the textual similarity and do not give stronger preference to complicated and costly tests. Therefore, we expect that IR techniques potentially perform better when hybridized with QTF. We also consider hybridizing with HIS to consider historical failure frequencies as well.

Before we hybridize OptIR to consider test execution cost or historical failure frequencies, we first consider a hybrid technique combining just test execution cost and historical failures. For a test T_i, given its historical failure frequency hf_i and its prior execution time t'_i, we define a TCP technique that rearranges tests by the descending order of:

$$a_{cch}(T_i) = hf_i/t'_i \tag{1}$$

We denote this technique as **CCH**. We use CCH as another comparison against the hybrid techniques that directly improve OptIR.

We design the first Cost-Cognizant IR-based TCP technique that orders tests based on their execution cost and IR score. For a test T_i, given its IR score ir_i (computed using the same optimized configuration values as with OptIR) and its execution time t'_i in the corresponding previous job, we define a TCP technique that rearranges tests by the descending order of:

$$a_{ccir}(T_i) = ir_i/t'_i \tag{2}$$

We denote this technique as **CCIR**.

We next consider a hybrid technique that combines OptIR with HIS. For a test T_i, given its IR score ir_i and its historical failure frequency hf_i, we define a TCP technique that rearranges tests by the descending order of:

$$a_{hir}(T_i) = (hf_i * ir_i) \tag{3}$$

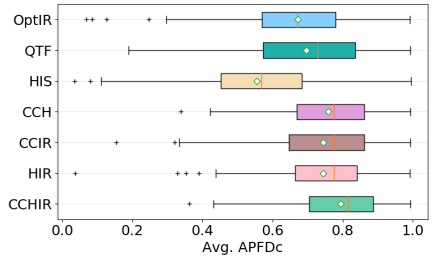
We denote this technique as **HIR**.

Finally, we consider a hybrid technique that combines both test execution cost and historical failures with OptIR. For a test T_i, given its IR score ir_i, its historical failure frequency hf_i, and its prior execution time t'_i, we define a TCP technique that rearranges tests by the descending order of:

$$a_{cchir}(T_i) = (hf_i * ir_i)/t'_i \tag{4}$$

We denote this technique as **CCHIR**.

Figure 3 shows our evaluation of the hybrid TCP techniques on the full dataset. From the figure, we see that all hybrid techniques are better than the individual non-hybrid techniques that they are built upon. For example, CCIR results in a better mean APFDc



TCP Technique	Avg. APFDc			
	mean	median	group	#B.Projs
OptIR	0.672	0.671	C	19
QTF	0.696	0.729	BC	9
HIS	0.556	0.568	D	5
CCH	0.758	0.776	AB	18
CCIR	0.744	0.776	AB	10
HIR	0.744	0.774	AB	31
CCHIR	0.794	0.815	A	47

Figure 3: Comparing against the hybrid TCP techniques

value than both OptIR (by 10.7%) and QTF (by 6.9%) from which this hybrid technique is constructed. Overall, we observe that the three-factor hybrid technique combining IR scores, prior execution time, and historical failure frequency outperforms any other technique we evaluate by at least 4.7% in terms of the mean averaged APFDc, which means the three factors we consider in the hybrid technique can enhance each other and form an effective hybrid technique.

To see whether there is a statistically significant difference between the distributions of the APFDc values of the different techniques, we once again perform a Tukey HSD test on the APFDc results for the three single techniques (OptIR, QTF, and HIS) and three hybrid techniques (CCIR, CCH, and CCHIR). We see a statistically significant difference between hybrid techniques and non-hybrid techniques, with CCHIR being the best and HIS being the worst. The results of these statistical tests suggest that the hybrid techniques, especially the three-factor hybrid technique CCHIR, are effective techniques to perform TCP.

Figure 4 shows the results of our per-project analysis comparing the different hybrid techniques. The figure shows the average APFDc of all jobs per project for each technique; the projects are sorted left to right by descending order of each one's average APFDc value for CCHIR. We do not include the per-project results of the non-hybrid techniques into the figure, because our prior overall evaluation on the full dataset shows that the non-hybrid techniques generally perform worse than the hybrid ones. The accompanying table in the figure shows the number of jobs and the average APFDc values across all those jobs per project for each hybrid technique. Once again, the best APFDc value is highlighted for each project.

Based on Figure 4, for most projects, CCHIR performs the best among all techniques or performs closely to the best technique, and it never performs the worst. Specifically, we perform a Tukey HSD test on the hybrid techniques on all 119 projects in our dataset that each has more than one job. We find that there are 105 projects for which CCHIR performs the best among all hybrid techniques, being in group A. Also, the standard deviation of the average APFDc per project for CCHIR is the lowest among all studied techniques, demonstrating that CCHIR tends to be more stable. These results further suggest that developers should in general use CCHIR.

We further inspect the projects that CCHIR does not perform as well. Among all the 27 projects that the average APFDc value is lower than 0.7 with CCHIR, there are 18 of them where HIR performs the best. We inspect these projects and find that these projects are those where QTF also performs poorly. Therefore, if CCHIR performs poorly on a project, it is likely due to the test cost factor. On closer inspection of these projects, we find that the failing tests are those that run quite long, so both QTF and the hybrid techniques that hybridize with QTF tend to prioritize those tests later. This finding further demonstrates the necessity of designing more advanced techniques to better balance the textual, cost and historical information for more powerful test prioritization.

> ***RQ3:*** *We find that hybrid techniques that combine OptIR with test execution cost and historical failures improves over OptIR with respect to APFDc, with CCHIR, which utilizes all these factors, performing the best among all hybrid techniques.*

5.4 RQ4: Impacts of Flaky Tests

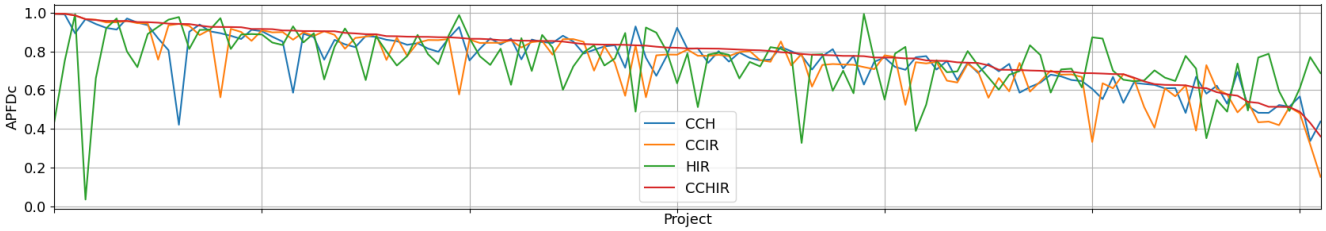
Figure 5 shows boxplots side-by-side comparing the APFDc for flaky jobs and non-flaky jobs under seven major TCP techniques (four change-aware IR techniques: Tf-idf, BM25, LSI, LDA; two change-unaware techniques QTF, HIS; and one hybrid technique CCHIR). We observe that all change-aware IR techniques (Tf-idf, BM25, LSI, LDA) perform better on non-flaky jobs than on flaky jobs. As flaky test failures are usually introduced into the program before the latest version [32] while non-flaky failures are more likely to be regressions introduced by the latest change, change-aware techniques are better at prioritizing non-flaky tests whose failures then likely indicate regressions in the program change.

We also find that QTF and HIS have better performance on flaky jobs than on non-flaky jobs. This result suggests that in our dataset flaky tests tend to run faster and are more likely to have failed in the past. Furthermore, CCHIR performs the best among the seven techniques on both flaky jobs and non-flaky jobs. This result demonstrates the robustness and effectiveness of our hybrid technique as it is positively influenced by the change-aware technique without suffering much from its weaknesses.

> ***RQ4:*** *Flaky test failures can affect the results from evaluating and comparing different TCP techniques. In particular, for change-aware techniques, these techniques would seem to perform better when evaluated using only non-flaky failures. However, we find that CCHIR still performs the best among all evaluated techniques regardless of flaky or non-flaky failures.*

5.5 Discussion

So far in our evaluation, we assume that each failure maps to a distinct fault. However, in reality it is possible that multiple test failures are due to the same fault in the source code, and with different failure-to-fault mappings the computed APFD and APFDc



Proj. Name	#Jobs	Avg. APFDc				Proj. Name	#Jobs	Avg. APFDc			
		CCH	CCIR	HIR	CCHIR			CCH	CCIR	HIR	CCHIR
zeroturnaround/zt-zip	3	0.994	0.994	0.439	0.994	flaxsearch/luwak	3	0.990	0.993	0.751	0.993
RIPE-NCC/whois	5	0.893	0.985	0.991	0.985	yandex-qatools/postgresql-embedded	2	0.966	0.966	0.037	0.966
mp911de/logstash-gelf	6	0.941	0.963	0.662	0.961	undera/jmeter-plugins	10	0.921	0.949	0.919	0.957
mitreid-connect/OpenID-Connect-Java-Spring-Server	5	0.912	0.950	0.969	0.957	apache/commons-compress	4	0.969	0.956	0.799	0.956
internetarchive/heritrix3	15	0.948	0.945	0.719	0.951	apache/incubator-dubbo	82	0.935	0.944	0.889	0.950
perwendel/spark	9	0.867	0.757	0.927	0.948	wmixvideo/nfe	20	0.806	0.934	0.963	0.943
mjiderhamn/classloader-leak-prevention	1	0.422	0.941	0.976	0.941	apache/systemml	6	0.901	0.931	0.812	0.941
eclipse-vertx/vert.x	95	0.938	0.884	0.910	0.933	vert-x3/vertx-web	8	0.903	0.912	0.912	0.926
ocpsoft/rewrite	42	0.893	0.564	0.971	0.926	graphhopper/jsprit	20	0.880	0.916	0.812	0.925
ebean-orm/ebean	44	0.864	0.900	0.886	0.924	xetorthio/jedis	67	0.911	0.854	0.889	0.915
DiUS/java-faker	16	0.904	0.908	0.887	0.915	alibaba/fastjson	46	0.875	0.898	0.845	0.914
FasterXML/jackson-databind	44	0.850	0.899	0.833	0.908	HubSpot/jinjava	7	0.587	0.860	0.929	0.907
killme2008/aviator	4	0.891	0.899	0.846	0.904	resteasy/Resteasy	101	0.873	0.879	0.893	0.904
rhwayfun/spring-boot-learning-examples	4	0.757	0.904	0.656	0.904	protegeproject/protege	3	0.859	0.885	0.824	0.903
alipay/sofa-rpc	72	0.836	0.813	0.918	0.895	gresrun/jesque	9	0.821	0.869	0.836	0.892
amzn/ion-java	11	0.879	0.875	0.652	0.888	dooApp/FXForm2	10	0.874	0.888	0.877	0.888
Angel-ML/angel	6	0.860	0.756	0.799	0.878	google/error-prone	26	0.853	0.871	0.727	0.878
prometheus/client_java	21	0.834	0.776	0.776	0.876	socketio/socket.io-client-java	4	0.842	0.844	0.885	0.875
orbit/orbit	5	0.814	0.858	0.786	0.875	openmrs/openmrs-core	48	0.798	0.858	0.733	0.873
ff4j/ff4j	9	0.868	0.863	0.883	0.872	yegor256/rultor	13	0.925	0.578	0.987	0.870
joelittlejohn/jsonschema2pojo	9	0.753	0.861	0.868	0.865	basho/riak-java-client	39	0.813	0.843	0.774	0.864
JSQLParser/JSqlParser	34	0.866	0.844	0.730	0.862	elasticjob/elastic-job-lite	86	0.836	0.844	0.813	0.858
LiveRamp/hank	7	0.866	0.857	0.628	0.857	bootique/bootique	5	0.759	0.820	0.868	0.855
RoaringBitmap/RoaringBitmap	44	0.861	0.847	0.699	0.855	tcurdt/jdeb	4	0.847	0.853	0.884	0.853
junkdog/artemis-odb	12	0.843	0.783	0.820	0.853	lukas-krecan/JsonUnit	57	0.880	0.863	0.602	0.850
zendesk/maxwell	27	0.850	0.863	0.722	0.844	FasterXML/jackson-dataformat-xml	7	0.788	0.849	0.795	0.838
apache/incubator-druid	93	0.808	0.701	0.831	0.836	pf4j/pf4j	13	0.825	0.827	0.727	0.835
hs-web/hsweb-framework	25	0.832	0.740	0.763	0.834	apache/servicecomb-pack	35	0.716	0.571	0.894	0.833
SpigotMC/BungeeCord	1	0.928	0.831	0.489	0.831	davidmoten/rxjava-extras	12	0.767	0.563	0.923	0.829
stanford-futuredata/macrobase	16	0.674	0.779	0.896	0.824	debezium/debezium	61	0.776	0.784	0.801	0.820
alibaba/druid	5	0.921	0.783	0.635	0.818	eclipse/paho.mqtt.java	20	0.813	0.809	0.789	0.815
jtablesaw/tablesaw	34	0.815	0.777	0.513	0.815	google/closure-compiler	38	0.741	0.777	0.783	0.814
scobal/seyren	6	0.801	0.782	0.793	0.814	twitter/GraphJet	13	0.747	0.775	0.782	0.811
weibocom/motan	24	0.794	0.796	0.661	0.809	abel533/Mapper	12	0.766	0.802	0.746	0.807
rest-assured/rest-assured	7	0.753	0.754	0.722	0.805	keycloak/keycloak	52	0.757	0.746	0.822	0.802
apache/incubator-dubbo-spring-boot-project	20	0.822	0.851	0.813	0.795	st-js/st-js	11	0.801	0.727	0.786	0.793
cglib/cglib	1	0.778	0.787	0.328	0.787	gchq/Gaffer	41	0.706	0.618	0.783	0.783
rapidoid/rapidoid	15	0.775	0.730	0.785	0.781	spring-projects/spring-data-redis	41	0.811	0.735	0.597	0.780
aws/aws-sdk-java	56	0.712	0.732	0.700	0.777	RipMeApp/ripme	74	0.777	0.731	0.584	0.777
qos-ch/logback	1	0.629	0.719	0.992	0.775	floodlight/floodlight	20	0.746	0.708	0.757	0.768
zhang-rf/mybatis-boost	19	0.768	0.779	0.551	0.768	ctripcorp/apollo	31	0.720	0.771	0.788	0.768
sakaiproject/sakai	39	0.705	0.524	0.823	0.764	awslabs/amazon-kinesis-client	28	0.769	0.744	0.390	0.764
fakereplace/fakereplace	2	0.775	0.738	0.525	0.753	alibaba/jetcache	4	0.706	0.746	0.754	0.751
spring-projects/spring-data-mongodb	41	0.750	0.649	0.692	0.750	networknt/light-4j	25	0.651	0.640	0.697	0.743
ModeShape/modeshape	7	0.742	0.736	0.801	0.742	apache/incubator-pinot	52	0.689	0.696	0.728	0.740
searchbox-io/Jest	13	0.736	0.561	0.666	0.728	JanusGraph/janusgraph	144	0.697	0.662	0.602	0.705
AxonFramework/AxonFramework	48	0.736	0.594	0.679	0.705	jhy/jsoup	25	0.587	0.740	0.699	0.704
google/auto	13	0.616	0.593	0.831	0.701	spring-projects/spring-security-oauth	27	0.639	0.645	0.781	0.700
vipshop/vjtools	21	0.680	0.701	0.588	0.696	teamed/qulice	12	0.671	0.680	0.707	0.695
getheimdall/heimdall	15	0.652	0.681	0.711	0.695	winder/Universal-G-Code-Sender	11	0.647	0.668	0.615	0.688
jcabi/jcabi-github	4	0.607	0.333	0.872	0.687	rickfast/consul-client	9	0.554	0.636	0.866	0.685
apache/rocketmq	49	0.668	0.610	0.700	0.683	demoiselle/framework	6	0.535	0.682	0.654	0.682
pippo-java/pippo	11	0.638	0.662	0.646	0.665	vipshop/Saturn	42	0.632	0.514	0.650	0.645
apache/zeppelin	37	0.627	0.407	0.702	0.631	onelogin/java-saml	4	0.609	0.609	0.665	0.627
codelibs/fess	45	0.611	0.568	0.647	0.627	square/moshi	5	0.483	0.625	0.776	0.625
alipay/sofa-bolt	20	0.669	0.392	0.711	0.613	alexxiyang/shiro-redis	8	0.582	0.729	0.353	0.610
doanduyhai/Achilles	3	0.623	0.606	0.550	0.590	pholser/junit-quickcheck	3	0.530	0.579	0.489	0.579
redpen-cc/redpen	16	0.694	0.486	0.737	0.571	jaeksoft/opensearchserver	6	0.519	0.540	0.495	0.540
apache/servicecomb-java-chassis	49	0.484	0.434	0.768	0.534	magefree/mage	43	0.483	0.438	0.787	0.514
spring-projects/spring-data-cassandra	40	0.523	0.420	0.595	0.514	shrinkwrap/resolver	3	0.515	0.513	0.493	0.513
opensagres/xdocreport	6	0.568	0.481	0.610	0.489	jenkinsci/java-client-api	3	0.338	0.320	0.770	0.431
sismics/reader	7	0.439	0.154	0.688	0.362						

Figure 4: Comparing the hybrid TCP techniques by project

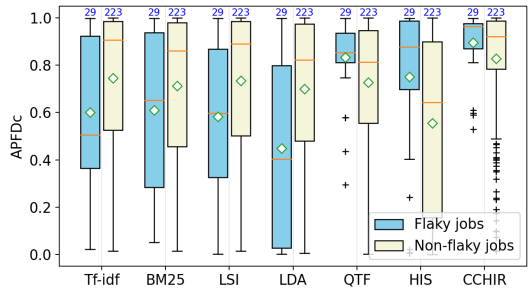


Figure 5: Comparing different techniques on flaky/non-flaky jobs

values could change. We evaluate again the major research questions by mapping all failures to the same fault to check whether the conclusions still remain the same. For RQ1, we calculate the mean and median APFD by job for Tf-idf, BM25, LSI and LDA; for RQ2, we calculate the the mean and median APFD and APFDc by project for OptIR, QTF, HIS, and the coverage-based techniques (only on the jobs in common among all the techniques); and for RQ3, we calculate the mean and median APFDc by project for OptIR, QTF, CCIR, HIS, CCH, CCHIR. For all these values, we recalculate while considering all failures mapping to the same fault. In other words, all we want to observe is how well the different TCP techniques order tests such to get a failed test to come as early as possible. We do not re-evaluate RQ4, concerning flaky tests, because our evaluation for RQ4 already only considers jobs with one test failure.

We compare the results with mapping each failure to a distinct fault (the type of evaluation we use before for all prior RQs) to see if there is a difference in overall results. Table 4 shows the evaluation results. From the table, we observe that although the APFD and APFDc values are larger when mapping all failures to one fault than when mapping each failure to distinct faults, different failure-to-fault mappings lead to similar overall conclusions.

The only result inconsistent with what we observe from before is that, after mapping all failures to the same fault, Additional Test Prioritization and Search-Based Test Prioritization outperform OptIR when evaluated by mean APFD. The potential reason is that both Additional Test Prioritization and Search-Based Test Prioritization care only about the additional coverage. Therefore, when there are multiple failures mapping to the same fault, the corresponding fault-revealing methods will grant only one of the failures a high priority. That is, the APFD and APFDc values are more under-estimated in these techniques than in other techniques. However, in terms of APFDc, the cost-aware metric, OptIR still outperforms the coverage-based techniques. Overall, all other conclusions concerning trends remain the same, i.e., BM25 is the best retrieval model for IR-based TCP, QTF outperforms all other non-hybrid techniques based on APFDc, and the hybrid CCHIR technique performs the best among all the hybrid and non-hybrid techniques.

6 THREATS TO VALIDITY

For our dataset, we extract test FQNs, test outcomes, and test execution times from job logs, so whether we can get a complete and accurate dataset is dependent on the completeness and the parsability of log files.

Our implementations of the TCP techniques and our experimental code may have bugs that affect our results. To mitigate this threat, we utilize well-developed and well-used tools, such as TravisTorrent to parse job logs, OpenClover to collect code coverage, javalang [60] to parse the AST of source code, and various mature libraries to compute IR similarity scores. Furthermore, we tested our implementations and scripts on small examples to increase confidence in the correctness of our results.

For our hybrid techniques, we collect the historical failures from the failed jobs in our dataset, which satisfy our requirements as discussed in Section 3, as opposed to failures from all possible failed jobs on Travis CI. As such, the historical failures we use is a subset of all possible failures for tests as recorded in Travis CI, which is in turn a subset of all possible failures that occurred during development (failures occurred during local development and not pushed for continuous integration). However, developers may not need to see *all* failures, as a recent study on evaluating test selection algorithms at Google focused on only the transition of test statuses (e.g., going from pass to fail), instead of just straight failures (developers may not have addressed the fault yet) [29]. We also construct our dataset using only jobs where the currently failing tests were not failing in the job before (Section 3).

Table 4: Impact of different failure-to-fault mappings

RQ No.		*One to One*		*All to One*	
RQ1	**APFD**	**mean**	**median**	**mean**	**median**
	Tf-idf	0.728	0.812	0.771	0.885
	BM25	0.752	0.844	0.795	0.913
	LSI	0.707	0.775	0.753	0.860
	LDA	0.659	0.735	0.713	0.837
RQ2	**Avg. APFD**	**mean**	**median**	**mean**	**median**
	OptIR	0.740	0.807	0.768	0.838
	QTF	0.393	0.305	0.440	0.397
	HIS	0.606	0.610	0.650	0.647
	Tot.	0.719	0.734	0.756	0.785
	Add.	0.725	0.740	0.774	0.802
	Sea.	0.722	0.746	0.771	0.796
	ARP	0.580	0.555	0.633	0.603
	Avg. APFDc	**mean**	**median**	**mean**	**median**
	OptIR	0.656	0.711	0.692	0.770
	QTF	0.701	0.768	0.733	0.806
	HIS	0.528	0.532	0.578	0.584
	Tot.	0.573	0.601	0.622	0.678
	Add.	0.582	0.595	0.637	0.672
	Sea.	0.578	0.564	0.634	0.665
	ARP	0.595	0.584	0.649	0.673
RQ3	**Avg. APFDc**	**mean**	**median**	**mean**	**median**
	OptIR	0.672	0.671	0.727	0.730
	QTF	0.696	0.729	0.733	0.778
	HIS	0.556	0.568	0.606	0.635
	CCH	0.758	0.776	0.793	0.826
	CCIR	0.744	0.776	0.779	0.804
	HIR	0.744	0.774	0.790	0.816
	CCHIR	0.794	0.815	0.826	0.853

The metrics for evaluating TCP techniques are also crucial for this work. We adopt both the widely used cost-unaware metric APFD and the cost-aware metric APFDc for evaluating all the studied TCP techniques.

7 RELATED WORK

Regression testing [64] has been extensively studied in the literature [13, 16, 17, 28, 49, 56, 57, 65]. Among various regression testing techniques, the basic idea of test-case prioritization (TCP) techniques is to prioritize tests that have a higher likelihood of detecting bugs to run first. Most prior work has implemented techniques based on test coverage (prioritizing tests that cover more) and diversity (ordering tests such that similar tests in terms of coverage are ordered later), and investigated characteristics that can affect TCP effectiveness such as test granularity or number of versions from original point of prioritization [19, 21, 30, 31, 33, 66]. Recent work shows that traditional coverage-based techniques are not cost effective at running the tests that detect bugs earlier, because they tend to execute long-running tests first [10]. To address this problem, one solution is to make new Cost-Cognizant coverage-based techniques that are aware of both the coverage and cost of tests [10, 36]. However, Chen *et al.* [10] showed that the Cost-Cognizant Additional Test Prioritization technique performed better than the cost-only technique for only 54.0% of their studied projects, and the mean APFDc of the Cost-Cognizant Additional Test Prioritization technique technique is just 3.0% better than that of the cost-only technique. Another solution is to utilize machine learning techniques, but it requires non-trivial training process involving a large number of historical bugs that are often unavailable [10]. In our work, we propose new simplistic hybrid TCP techniques that combine an IR-based TCP technique with test execution cost and historical failure frequency. We find that these hybrid techniques perform better, leading to higher APFDc values.

Prior work has utilized information retrieval (IR) techniques in software testing [24, 41, 53]. Our work is most similar to prior work by Saha *et al.*, who prioritized tests in Java projects using IR techniques, ordering tests based on the similarities between tests and the program changes between two software versions [53]. Using IR to perform TCP can be effective, because it does not require costly static or dynamic analysis, only needing lightweight textual-based computations. Saha *et al.* found that their IR-based TCP technique outperforms the traditional coverage-based techniques. However, Saha *et al.* evaluated their technique on a dataset with only 24 manually-constructed version pairs from eight projects to reproduce regressions, which potentially makes their result noninclusive and less generalizable. Also, they evaluate using APFD, which does not consider the cost of testing and does not reflect the effectiveness of TCP as well as using APFDc. Our work improves upon Saha *et al.*'s evaluation by introducing a much larger dataset with real failures and real time, and we utilize APFDc to evaluate the effectiveness of IR-based TCP techniques. Mattis and Hirschfeld also recently studied IR-based TCP on Python code and tests [38]. They propose a new IR-based technique based on the predictive power of lexical features in predicting future failures. Mattis and Hirschfeld find that their predictive model outperforms BM25 based on APFD measured through seeded mutants and faults in the dynamically-typed programming domain of Python, but their new technique does not outperform BM25 when time to first failure is considered.

Recently, some other work has used Travis CI to construct large and real datasets to evaluate testing techniques [8, 20, 25, 39, 56]. We also collect our dataset from Travis CI, collecting information from builds that involve real changes from developers and real test failures. With real failures used in evaluation, our work improves upon evaluation of TCP in prior work, which utilized mutation testing [7, 10, 23]. Mattis *et al.* in a parallel, independent work published a dataset for evaluating TCP [6, 39]. For their dataset, Mattis *et al.* also collected test failures from Travis CI, collecting more than 100,000 build jobs across 20 open-source Java projects from GitHub. Their dataset contains the test outcomes and test times per job, but it does not contain the source code diffs we need to perform IR-based TCP.

There has been much prior work in the area of flaky tests [8, 18, 26, 27, 32, 55, 58]. In our work, we evaluate how TCP techniques perform on flaky and non-flaky tests separately. From our evaluation, we find that the change-aware IR-based TCP techniques perform better when considering only non-flaky test failures. However, we find that the hybrid TCP technique CCHIR is quite robust in the face of flaky tests.

8 CONCLUSIONS

In this work, we conduct an empirical evaluation of TCP techniques on a large-scale dataset with real software evolution and real failures. We focus on evaluating IR-based TCP techniques, and our results show that IR-based TCP is significantly better than traditional coverage-based techniques, and its effectiveness is close to that of the cost-only technique when evaluated by APFDc, a cost-aware metric. We also implement new hybrid techniques that combine change-aware IR with historical test failure frequencies and test execution time, and our hybrid techniques significantly outperform all the non-hybrid techniques evaluated in this study. Lastly, we show that flaky tests have a substantial impact on the change-aware TCP techniques.

ACKNOWLEDGMENTS

We thank Wing Lam, Owolabi Legunsen, and Darko Marinov for their extensive discussion with us on this work. We acknowledge support for research on test quality from Futurewei. We also acknowlege support from Alibaba. This work was partially supported by NSF grants CNS-1646305, CCF-1763906, and OAC-1839010.

REFERENCES

[1] [n.d.]. GitHub. https://github.com/.
[2] [n.d.]. Maven. http://maven.apache.org/.
[3] [n.d.]. Travis-CI. https://travis-ci.org/.
[4] 2020. Empirically Revisiting and Enhancing IR-based Test-Case Prioritization. https://sites.google.com/view/ir-based-tcp.
[5] 2020. Maven Surefire Plugin - surefire:test. https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html.
[6] 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. https://zenodo.org/record/3712290.
[7] Maral Azizi and Hyunsook Do. 2018. ReTEST: A cost effective test case selection technique for modern software development. In *ISSRE*.
[8] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*.

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *MSR*.

[10] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *ESEC/FSE*.

[11] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *JASIS* 41, 6 (1990).

[12] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*.

[13] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *TSE* 28, 2 (2002).

[14] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *FSE*.

[15] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *ISSTA*.

[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *ICSE Demo*.

[17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*.

[18] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*.

[19] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *ICSE*.

[20] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *ASE*.

[21] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *ASE*.

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA Demo*.

[23] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*.

[24] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. 2014. Test case prioritization based on information retrieval concepts. In *APSEC*, Vol. 1.

[25] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*.

[26] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*.

[27] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.

[28] 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 *FSE*.

[29] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing transition-based test selection algorithms at Google. In *ICSE-SEIP*.

[30] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search algorithms for regression test case prioritization. *TSE* 33, 4 (2007).

[31] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *ICSE*.

[32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.

[33] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *FSE*.

[34] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. 2018. Assessing test case prioritization on real faults and mutants. In *ICSME*.

[35] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2018. Predictive test selection. In *ICSE-SEIP*.

[36] Alexey G. Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-cognizant test case prioritization.* Technical Report. Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln.

[37] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2010. Introduction to information retrieval. *Nat. Lang. Eng.* 16, 1 (2010).

[38] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *Programming Journal* 4 (2020).

[39] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An open-source dataset for evaluating regression test prioritization. In *MSR*.

[40] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE-SEIP*.

[41] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 2011. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *ICWS*.

[42] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE*.

[43] Marek Parfianowicz and Grzegorz Lewandowski. 2017–2018. OpenClover. https://openclover.org.

[44] David Paterson, José Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An empirical study on the use of defect prediction for test case prioritization. In *ICST*.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011).

[46] Radim Řehůřek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. In *LREC*.

[47] Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Ret* 3, 4 (2009).

[48] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. 2000. Experimentation as a way of life: Okapi at TREC. *Inf. Process. Manage.* 36, 1 (2000).

[49] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *TSE* 22, 8 (1996).

[50] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *ICSM*.

[51] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *TSE* 27, 10 (2001).

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

[53] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *ICSE*.

[54] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.* 24, 5 (1988).

[55] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*.

[56] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*.

[57] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *PACMPL* 3, OOPSLA (2019).

[58] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*.

[59] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA*.

[60] Chris Thunes. 2018. c2nes/javalang. https://github.com/c2nes/javalang.

[61] John W. Tukey. 1949. Comparing individual means in the analysis of variance. *Biometrics* 5, 2 (1949).

[62] Xing Wei and W. Bruce Croft. 2006. LDA-based document models for ad-hoc retrieval. In *SIGIR*.

[63] Frank Wilcoxon. 1945. Individual comparisons by ranking methods.

[64] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *STVR* 22, 2 (2012).

[65] Lingming Zhang. 2018. Hybrid regression test selection. In *ICSE*.

[66] 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*.

[67] 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*.