Evaluating Information Retrieval for the use in Regression Test Selection

Case Study

Author:	Majd Akleh
Supervisors:	**Prof. Dr. Ben Hermann**
	TU Dortmund
	Raphael Nömmer
	CQSE GmbH
Date:	September 2023

Contents

1 Introduction

During the software development life-cycle, code evolves by fixing bugs and introducing new features, however, these enhancements usually come along with potential problems, and might negatively impact existing functionalities. Testing is a crucial way to handle these problems by identifying issues early in the development process and ensuring that software or products meet quality standards before they reach the end-users. Regression testing is a type of software testing that focuses on verifying that changes or modifications in a software application do not introduce new defects or regressions in existing or newly introduced functionality. It involves retesting previously tested features or functionalities to ensure that they still work correctly after changes have been made to the software. The primary objective of regression testing is to identify and catch any unintended side effects or issues caused by modifications to the software. It helps ensure that the system remains stable and reliable throughout the development life-cycle, even as new features are added or bugs are fixed.

1.1 Test Selection

Modern software applications can have extensive test suites comprising hundreds to tens of thousands of test cases. And as new features are added, more test cases need to be created. Besides, old tests rarely get removed, which cause the test suite to grow dramatically in size. Executing all of these test cases can require a significant amount of time, especially if each test case takes a non-trivial amount of time to run, i.e. tests at higher levels of abstrac-

tion, such as system or UI tests are usually time consuming, with individual tests often requiring several minutes or even longer to complete. This has a significant impact on the progress of developers and the overall development process, i.e. waiting for tests to complete before getting feedback can result in downtime where developers are unable to proceed with their work, this also causes them to switch context while waiting for tests to finish, which in turn can reduce efficiency and focus. Furthermore, running a large number of test cases simultaneously can consume substantial computing resources such as CPU, memory, and disk space. This can lead to resource contention and slower execution times for individual test cases. Therefore, test selection techniques are used. Test selection is a software testing technique that aims to optimize the testing process by selecting a subset of test cases from a larger pool of available tests. This aims to optimize the testing process by selecting a subset of test cases that have the highest likelihood of detecting defects while minimizing the time and resources required for testing, which in turn can accelerate the overall development cycle. Several popular testing selection techniques are used in software development, such as Code Coverage-Based Selection or Impact Analysis.

1.2 Motivation for Information Retrieval

Numerous selection methodologies commonly employed in both literature and practical applications rely on the concept of code coverage. However, as the size and intricacy of projects grow, gathering and assessing code coverage becomes progressively challenging. This gives rise to a situation where these methodologies tend to exhibit limited scalability when applied to projects

of considerable size. Therefore, this study aims to develop and compare a prototype procedure that does not rely on coverage with existing coverage-based methods. Instead Information retrieval (IR) techniques are to be used and investigated, these have been shown to be effective for selecting and prioritizing tests. IR deals with the organization, retrieval, and analysis of information from large collections of data, typically in the form of text documents. Reducing the problem of test selection into the traditional IR problem allows us to benefit from the massive research progress in this field. In this study, we are going to have a deeper look into how test selection can be formulated using an IR algorithm, the design will take into consideration textual distance between tests and changed source code. Next, we will realize the conceptual principles into functional implementation where we investigate different parameters. Finally, we put our design under evaluation based on historical failure patterns which are grabbed from an open-source and industrial contexts.

Now the question: Is our design more effective than untreated or random test orders? This research question aims to understand whether our algorithm reveals regression faults earlier than when there is no test selection or when test cases are ordered at random.

1.3 Study Structure and Objectives

In Section 2, we provide a literature review of IR algorithms and test selection, discussing previous studies and related work. Section 3 details the methodology employed for conducting the case study. We explain the criteria used for selecting specific algorithms and techniques. Following the

methodology. Section 4 presents the results and analysis obtained from the case study. In Section 5, we engage in a comprehensive discussion of the results, comparing and contrasting the performance of the algorithms. Next, in Section 6, we address the limitations encountered during the case study. We openly discuss any constraints or biases that may have influenced the results. In Section 7, we draw conclusions based on the key findings and insights obtained from the case study. Finally, the report concludes with a comprehensive list of references in Section 8, ensuring the proper acknowledgment of all cited sources.

2 Literature Review

Information Retrieval (IR) plays a vital role through its implementation in a range of practical applications, including searching the web, question answering systems, personal assistants, chatbots, and digital libraries among others. The primary objective of IR is to locate and retrieve information that is relevant to a user's query. As multiple records may be relevant, the results are often ranked according to their relevance score to the user's query [1]. Information retrieval techniques have also helped to improve the efficiency and effectiveness of software development, for example, Liu et al. [2] used an IR method called Latent Semantic Indexing (LSI) to store method information extracted from the source code and method execution trace, in which developers can query this to locate features in the codebase, this can be helpful for developers who are trying to learn about a new feature or fix a bug. Latent Dirichlet allocation, a generative statistical model that has sig-

nificant advantages over LSI, was also investigated in similar contexts. An LDA-based technique was proposed for automatic bug localization, which showed sufficient accuracy across different test projects [3].

The initial proposal that suggests employing Information Retrieval (IR) techniques for the purpose of test case selection and prioritization was published by Saha et al. [4], where the idea was that test cases exhibiting greater textual similarity to the modified code are more likely to be associated with the changes made to the program.

3 Methodology

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 [5].

The idea behind our approach is that the program changes between revisions are a good indication of the areas of the code that have been modified and this can guide towards the tests that are most likely to be affected by the changes. Now by treating these changes as a query, the IR system can identify a set of tests that has a high relevance and textually more related with the provided query, and prioritize them according to some relevance score.

Therefore, as initial step, we collect test files from their corresponding test

directories for a given software project, process each test including tokenization and filtering, then construct and index a so-called document collection, that represents the pool of tests to be queried from. At this point, the program is ready to accept changed files, tokenize and filter them almost in the same way as test documents, and construct the final query that combines information for the entire set of these changed source files. That basically summarizes our focus into two major program components, the **Indexer**, i.e. the construction of the document collection, and the **Searcher**, i.e. the system that query these indexed documents. We are going to discuss these two components in detail in the next sections.

3.1 Lucene

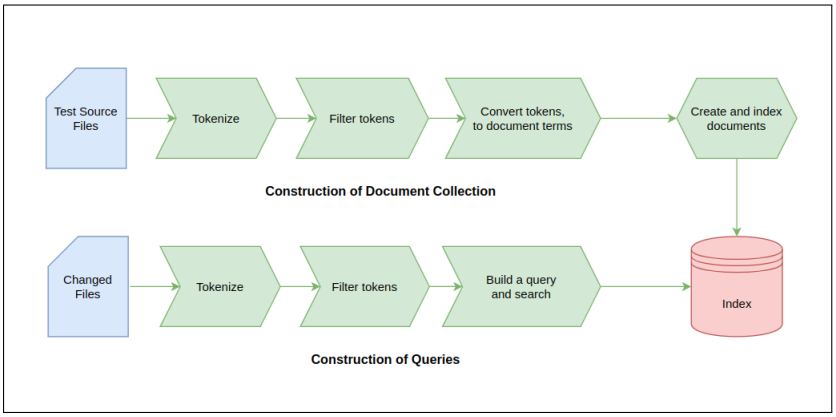

Figure 1: Algorithm Schema

We adopt the Lucene library [6], an efficient open-source search engine library written in Java. We decided on Lucene since it provides powerful indexing and retrieval capabilities. Lucene operates on the concept of an

inverted index, which is a data structure that maps terms to the documents in which they appear. We are going to use it to analyze documents, and builds an index that allows for efficient full-text searching. The index can handle large volumes of data and provides fast retrieval of relevant documents based on search queries.

3.2 Construction of Document Collection

In the context of IR-based test selection, the data objects originate from the individual tests. A test file in this context is an entire file that might include one more test classes. Each of these classes might in turn include a couple of test methods. The aim here is to construct one document for each test file and not for classes or methods, i.e. the test selection will return test file names or paths that could be run individually as a test unit. We decided on this approach as a test file might include not only the test method which most likely to identify the bug in the revision, but also other related test methods or units that also share some context with changed source files as well. This granularity will provide a higher coverage level

3.2.1 Collecting Test Files:

In order to build the test files index container, test files in the corresponding software system can be collected from all resources available. This typically involves identifying the relevant files and gathering them into a single location. Manual search within the software's directory or repository could be performed to locate the test files. Folders or directories specifically dedicated to testing can be looked, such as "tests," "test," or any other naming

conventions related to testing. Build or test automation tools could also be used if the software uses them, such as build systems like Maven or testing frameworks like JUnit, we can utilize their functionalities to gather the test files. These tools often have specific directories or settings that store the test files. In this study, we collect test files by conducting a file search on a given root directory, test directory can be chosen directly instead of the project root in order to narrow the search scope. Currently we also only support test files that are written in Java or JavaScript, therefore we have to filter the returned files obtained from the search to fit this restriction as well.

3.3 Tokenization

Given that we are working with program source code instead of English natural language, our tokenization process differs from standard IR systems used outside the scope of software engineering. In a standard IR tokenization task, the text to be indexed is usually separated using the white space tokenizer. Although, this approach is simple and straightforward, but it does not filter out meaningless terms for IR such as Java keywords (e.g., if, else, return), operators, numbers and open-source licenses[5]. However, in the context of test files, tokenization involves breaking down the source code into elements or tokens. Tokens can represent various elements in the code, such as keywords, identifiers, literals, and symbols. This is a crucial step for the analysis process, as it provides the flexibility to extract the information we need, that will be most related to the queried changed files later.

For this, we use **Teamscale** [7] that provides an API which is capable of parsing source files on a wide range of programming languages and is responsible

for breaking down the source code into individual tokens, including entities like classes, methods and statements.

3.3.1 Token Filtering:

Our approach tries to index as much context information as possible, in other words, information that is highly relevant and directly contributes to understanding the textual relationship between a source and the corresponding test file and provide more meaning similarity such as describing the same area or UI components in the project, this would be considered most useful for indexing and analyzing. Therefore, tokens representing the programming language keywords or syntax for example aren't considered very valuable in determining the context of the test file. Instead, we can extract various elements such as identifier names (class, attribute, method and variable names) as well as comments, string literals and descriptions. These hold significant value for information retrieval since they serve as locations where developers can employ their own natural language terms, and considered important to link contexts between the test and source files [4].

3.4 Convertion To Document Terms:

After we received our filtered tokens from the last step, we now pre-process these tokens to acquire the suitable input for creating documents. This pre-processing consists of multiple steps that are necessary to maintain a better document retrieval later, such as:

- **Camel case handling:** In general, camel case identifier names consist of concatenated words such as method or variable names, in which we

basically destruct into separate lower-case words. E.g *testHelloWorld* becomes *test*, *hello* and *world*

- **Stop Word Removal:** removes the common English language stop words, such as propositions, articles and question words.
- **Stemming:** this considered an important step in an IR system in order to reduce words to their base or root form. It involves removing prefixes, suffixes, and other variations from words so that different forms of the same word are treated as the same token.

This way we acquire a list of words or "terms" for each test file, this list contains the most needed information that represent that test file. Usually, the next step would be to index this list as one document, however we decided to split this list into three categories of terms based on the program constructs. We distinguish between:

- Method names
- Docs and comments
- All other identifier names (e.g. variable names, string literals)

3.5 Indexing

After parsing test files and converting them to terms and splitting these terms into three categories, the indexing process takes place. Indexing in this case means converting these text terms into a structured format that allows for efficient and effective searching. This allows for fast look-ups during search operations. Creating a document in Lucene allows to also create

several fields inside each document for organization, therefore, we use this to assign a field for each of the three categories mentioned previously. The aim behind this is to also set different weights for each category, we think that finding a term in the test method name has a higher potential to share context with the queried source file than finding the same term in an arbitrary identifier inside the method. Object return values such as *String*, *ArrayList* or *Descriptor* are also considered identifiers, however they contain less value since they are more likely to appear in many test files. However, these identifiers are also less likely to be included in the test method names or even comments. Considering all this, we decided to create three fields for each category and assign the highest scoring value for terms found in test method names, less scoring value for terms found in comments and docs, and the least value for the remaining terms. This scoring scheme will be considered when deciding on the returned documents. Figure [2] shows a typical test file on the left yet to be analyzed and indexed, and on the right we see its document representation divided into three fields as described above.

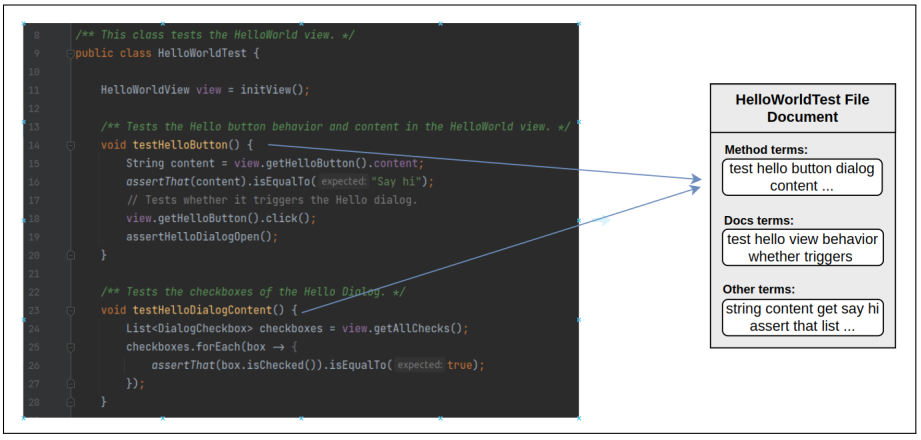

Figure 2: Conversion from test source file to document representation.

3.6 Query Construction

As we defined in the problem formulation, in an IR-based test selection, changes between two program versions comprise the query. Generally, there are multiple approaches to extract these differences from the changed program, for instance:

- **At line level:** which basically considers differences between two versions at the line level only. A disadvantage at this level is being too sensitive against tiny changes.
- **At a local level:** applied on wider area around the changes to include more context, such as a block or method level, which enhances the previous approach and tackles its weakness.
- **At a file level**: considers the entire changed file or class as a query after re-representing it as a custom structure by extracting as much useful information as possible.

In this study, we decided on the last approach as we believe the main advantage of it, is the ability to include more related contexts, such as unchanged methods or the import list, that could potentially hold extra information that could affect the test selection result dramatically. After acquiring the changes, we parse these set of files similar to how we processed test files before, i.e we tokenize, filter, and convert to query terms, except this time there is no need to differentiate between three categories of terms. Instead, we collect one list of terms extracted from all changed files and construct exactly one query where each of these terms may or may not appear in the resulting

document. We decided on this "one query" approach instead of querying each changed file separately, because the files that have been changed together in one commit have higher potential to share the same area and context, and we basically want this entire area to be tested after the change. This way, the test files that have higher similarity with this query are returned, and the tests that match more terms from the changed set would have higher score and thus higher priority. This also preferably assumes that test units/files are constructed in such a way, for each file, one area of the program is being tested, that has multiple related contexts.

4 Results and Analysis

To investigate the effectiveness of our method, we performed an empirical evaluation which aims to assess the practical application and outcomes of the proposed solution, by comparing it with a random test selection approach as baseline. In a random test selection, test cases are selected randomly from a test suite for execution, however, for evaluation purposes, we perform test selection sampling 20 times and average the accuracy results. We refer to the random approach by **Rand**. We conduct experiments on several project systems, and we use different configurations and implementations. With that we aim to answering the following research questions: (1): How do different IR configurations impact IR-based test selection techniques? (2): How can we further enhance these techniques?

4.1 Subject Systems

In our case study, Defects4J [8] serves as the experimental environment for our empirical evaluation and analysis. Defects4J is a database of diverse reproducible real-world software bugs from open-source Java projects. These bugs have been encountered in actual software development scenarios, making them highly relevant for our practical investigation.

Project ▲	Files	Source Lines of Code	File Size Assessment	Longest Method Length	Total Number of Bugs	Total Number of Tests
Summary	6.4k	928.6k		1k	825	2294
Chart	1.1k	149k		301	26	329
Cli	40	4.5k		97	39	29
Closure	992	260.7k		1k	174	241
Codec/src	43	7.1k		220	18	59
Collections/src	489	57.8k		207	4	206
Compress/src	81	8.7k		248	47	140
Csv/src	15	2.1k		55	16	20
Gson	258	17.4k		57	18	97
JacksonCore/src	128	21.9k		103	26	143
JacksonDatabind	635	75.8k		128	112	503
JacksonXml/src	117	10.5k		118	6	82
Jsoup/src	35	3.8k		81	93	49
JxPath/src	203	24.6k		371	22	54
Lang/src	232	56k		286	64	138
Math/src	795	90.9k		317	106	112
Mockito	931	56.8k		890	38	162
Time/src	315	80.9k		233	26	153

Table 1: Defects4J Projects Overview

Table 1 presents an overview of the projects provided by Defects4J, where we can see relevant metrics such as number of files, SLOC and file size. This information gives the ground knowledge about each project, that helps to draw conclusions about the correlations between their metrics and the overall

model performance. The number of files directly affects the size of the search space that needs to be considered when identifying tests, i.e. it can affect the efficiency of indexing and searching. Indexing a large number of files may require more computational resources and time. Additionally, searching across a larger number of files can impact the speed of the selection process. Although these metrics are taken for the entire project and not for the test directories only, SLOC still gives a detailed idea about the size of the index in general, since each line of code contributes to the overall size. The quality of the index can also be influenced by file size. A larger test file may introduce more noise into the index document, making it potentially less effective at retrieving relevant information if not properly managed. For that, the table shows a widget that break down file sizes into three categories, small files $\leq$ 300 SLOC in green, medium files $\leq$ 750 SLOC in yellow, and large files $>$ 750 SLOC in red. This might not only affect document size, but also query length dramatically, since we deal with queries that are basically lists of terms collected from a set of changed files, i.e. they might contain ten of thousands or terms. Defects4J provides a collection of bugs for each project, for each bug, a list of the modified classes are provided alongside with the relevant, and triggering tests which correspond to this change for a specific project. According to the authors, a test class is relevant if, when executed, the JVM loads at least one of the modified classes. A triggering test however, is the one that trigger (expose) the bug. We can use this information to investigate whether using these modified classes as input for our model would yield some of the corresponding true failing tests.

4.2 IR Configurations

One more objective of this study is to investigate, how do different information retrieval configurations impact the results of the proposed model. Subsequently, we will adopt the configuration that exhibits the highest performance as the default implementation. There are two main configuration options we need to set when implementing our approach.

- **Default:** Which was explained in the methodology in section 3, and we refer to it by IR.
- **Distinct terms:** Given that there is often a big difference between two program versions, such as thousands of lines of code, it is quite probable that they will contain numerous duplicated terms. Therefore we introduce a compact version by removing all the duplicated words. We refer to it by IRD.

4.3 Max Tests Returned

We tested our model on different values for the test percentage allowed to be returned from the search operation, ranging from 10% to 90%. Figure 3 depicts the relationship between the percentage of tests returned after querying and the overall accuracy of detecting tests, both triggering and relevant.

We see here how our IR and IRD models both show significant improvement over the random approach, with IR slightly superior to IRD. For better analysis, Figure 4 presents box plots representing the ranges in the previous

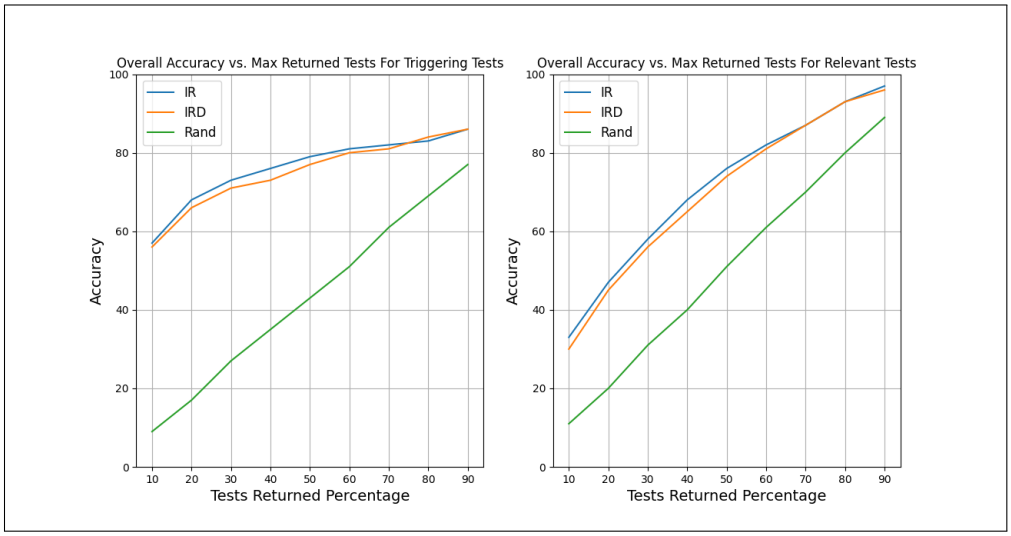

Figure 3: Accuracy rates for different max returned tests.

figure. These ranges provide a sense of the variation in the data distribution across all projects.

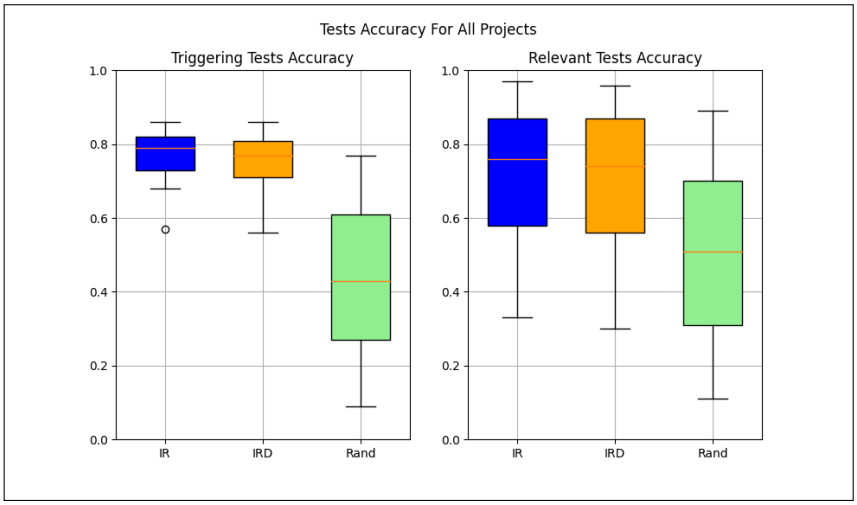

Figure 4: Triggering and Relevant Accuracy Box Plots

The range of values for IR / IRD triggering tests accuracy spans from

57% to 86% and from 56% to 86% respectively. However, the IQR* shows compact data results for both configurations with medians 79% and 77%. For relevant tests on the other hand, a more data spread is observed, accuracy here spans from 33% to 97% and from 30% to 96% respectively. We also observe that the data has less skewness and the medians tend towards the center.

*The interquartile range (IQR) is the box that contains the middle 50% of the data.

4.4 Results By Project:

Table 2 drills down the previous overall data by showing one data slice for one value of the metric of max returned tests, particularly 30%, it provides results of testing our default IR model against the projects provided by Defects4J and contains information, such as, average triggering test taken across all bugs in each project, triggering tests capture accuracy, and average query times taken for all bugs. This provides an indication how the overall accuracy is maintained in the previous figures. We witness accuracy ranges from 50% to 100% and 32% to 88% for triggering and relevant respectively. Query time was also present in the table, this shows the time needed for the algorithm to perform a query including parsing changed files, extracting terms, processing tokens and then search the index for relevant tests. However, these numbers are taken as average across all bugs in the corresponding project.

Project	**Avg. Change Size (KB)**	**Avg #Trig.**	**Trig. Acc.**	**Avg #Relev.**	**Relev. Acc.**	**Avg. Query Time**
[JacksonDatabind]	36	2	74%	260	40%	14.96 ms
[Csv]	23	2	82%	6	64%	5.88 ms
[Jsoup]	28	2	81%	16	51%	9.34 ms
[Cli]	16	2	58%	15	44%	4.95 ms
[Math]	27	2	88%	19	84%	12.03 ms
[Compress]	24	2	77%	21	50%	5.98 ms
[JacksonCore]	69	3	60%	53	46%	15.19 ms
[Codec]	28	3	78%	6	84%	5.89 ms
[Chart]	52	4	67%	36	79%	13.69 ms
[Mockito]	8	4	65%	126	48%	4.45 ms
[Collections]	40	1	100%	5	83%	10.25 ms
[Closure]	41	4	69%	87	49%	16.02 ms
[Lang]	58	2	88%	7	88%	9.03 ms
[JxPath]	17	2	55%	16	37%	6.23 ms
[Gson]	22	2	64%	49	51%	7.94 ms
[Time]	48	3	71%	73	53%	12.38 ms
[JacksonXml]	23	2	50%	51	32%	7.83 ms

Table 1: Default IR Accuracy for test return percentage of 30%

5 Evaluation

We saw in Figure 3 how IR algorithms beat the random guessing approach by a good margin. This shows that the textual relevance between the changed code and the corresponding test files plays a huge role in test selection and prioritization. Such relevance is not considered at all in the random approach that has no mechanism to re-order or select result tests. One more observation we also extract from Figure 3 is that IR performs slightly better in terms of accuracy than IRD, and this due to the algorithm how relevance is being calculated, especially that duplicate query terms may carry additional context or emphasis. By retaining duplicates, the searcher can consider the frequency and position of each term, potentially boosting the score of documents that contain multiple occurrences of a that term, providing more relevant results. However, keeping duplicates naturally increase the length of the query. Longer queries may require more computational resources for additional comparisons and processing. This extra processing overhead can slow down the search, especially when dealing with a large number of documents. Also when duplicates are present, index size can increase, which in turn can affect search speed.

In Table 2, data per project was presented. We saw how the model performed quite well on the triggering tests, however, this might be influenced by the the fact that the number of triggering tests per bug is quite low (avg. between 1 and 4 triggering tests per project). Therefore, returning 30% of the test suite has quite good chance of picking the triggering tests. This correlation appears most for projects with only 1 or 2 average triggering tests. Nevertheless, the model still outperform the random guessing algorithm even

for relatively high returned test percentage allowed. For relevant tests, the model has suffered for specific projects, we argue that this due to the number of relevant tests per project, as capturing more tests is more challenging. One more reason is that relevant tests differ from triggering test as they don't necessarily provide semantic context similar to the changed files, this could raise an obstacle for IR based techniques in general that highly depend on the textual similarity. Despite that, the model had better performance than the random guessing overall as we saw in Figure 3. Finally, the query time column gives data about the model performance for each project, this metric does not seem to have an interesting correlation with the other columns except for the change size, which was calculated as a sum of all changed file sizes in KB. This is due to the the fact of bigger files has more terms to process and in turn to query. Some of these files might have thousands of lines, that is the tendency among the development team towards writing smaller files that are dedicated for one purpose, and extracting the remaining logic into other files might have a positive impact, not only on the performance but also on the results.

6 Limitation

Our model rely on the the availability and quality of source/test code documentation, including well naming of variables and methods as well as meaningful comments that generally contain rich context. This dependency may affect test selection results when incomplete or missing. However, this might not be considered a big issue in practice, because in the majority of cases,

tests are named in alignment with the source code. Furthermore, it may lack the broader context in which code operates, i.e. code changes that affect high level abstractions or class generics provide less value in terms of textual context, which reduce accuracy. As mentioned in section 4, the model is designed to return full test files and not failing test methods in particular, also this assumes that test files contain related test methods, which might not be the case in practice. On other side, flickering tests showed to be a struggle for test selection methods that rely on code change textual context and have negative impact on the results. Our model does not handle these tests which might have misled our results. Regarding performance, we tested relatively small repositories with small test suites, however for large codebases, indexing and querying might become computational expensive, which limits the scalability of our model.

7 Further Enhancements

At the current state, the model considers the tokenized program changes as input, which we considered the main source of context between source and test files, however we can improve this input by including more features other than the textual difference, e.g author of the source code which might reflect information about the tests, since features are most likely accompanied with tests written by the same developer. We can proceed further and include more semantics related to the change such as the commit message, bug descriptions imported from the software planning environments, and more. These texts might add more values especially for tiny changes that is not

enough to identify related tests. Another enhancement might be by considering the historical data and version control system logs to allow the model better understand how changes in this area have impacted tests in the past. On the performance side, we can optimize the design to ensure that it can scale to handle large codebases and frequent changes, cloud-based solutions can be applied to handle these scalability challenges.

8 Conclusion

Regression testing is a crucial part of the software development life-cycle. It involves retesting existing features to make sure they still work correctly after changes or additions have been made to the software. The main goal is to catch any unintended issues caused by these modifications, ensuring the system remains stable and reliable as it evolves. Throughout this study, we presented a test selection approach based on an information retrieval method, We have examined the key components of the algorithm, its implementation, and its challenges. Next, we conducted an empirical evaluation of different configurations of the algorithm. For that we used a dataset of 17 projects that contain real bugs and their failing tests. Our results showed that the IR model is significantly better than a random test selection approach. In summary, our model has provided valuable insights into the effectiveness and applicability of this approach in the field of software testing. It showed that IR-based test selection can be a promising area for enhancing software testing, as the potential benefits in terms of efficiency, adaptability, and test suite relevance make it a valuable strategy for organizations striving to deliver

high-quality software products.

9 Appendix

9.1 Company Information

CQSE GmbH

Centa-Hafenbrädl-Straße 59

81249 München, Germany

Email: info@cqse.eu

CQSE GmbH, short for "Continues Quality Software Engineering," is a quality software consulting and solutions provider. Established in 2009, CQSE is headquartered in Munich, Germany. The company specializes in offering consulting services, software analysis, quality assurance, and improvement solutions to clients across various industries, that help customers to evaluate, improve and control the quality of their software systems.

9.2 Acquired Experience and Learning:

During my case study at CQSE, I had the privilege to work on large software applications mainly Teamscale (TS), a software intelligence platform that provides software analysis and quality assurances. Moreover, I was introduced to the realm of software testing, and the challenges that face companies in industry when testing their own large scale applications. I researched Information Retrieval (IR) techniques and its application in regression test selection. This phase involved identifying relevant literature and exploring

state-of-the-art IR methods. I also worked with professionals who guided me throughout the process of integrating my knowledge in the field of Data Science with their research ideas, in order to produce solutions that could potentially facilitate their decisions regarding testing, and improve the efficiency of their software testing processes in general. With the help of TS, I was able to implement my own approach of test selection, and contribute to CQSE's work goals. Furthermore, I put my hands on open-source projects, that served as the basis of my methodology on one hand, and used as data to test my approach on on the other hand. I also maintained detailed documentation throughout the case study, and provided regular progress reports to CQSE and my university. Finally, My study at CQSE was a informative experience that significantly expanded my knowledge with a comprehensive understanding of software quality, emphasizing the vital role of regression testing in maintaining high-quality software products. Collaboration and communication were key, as I learned the significance of effective teamwork and professional interaction while working with their team.

References

[1] K. A. Hambarde and H. Proenca, "Information retrieval: Recent advances and beyond," *arXiv preprint arXiv:2301.08801*, 2023.

[2] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the 22nd IEEE/ACM International Conference*

on Automated Software Engineering, ASE '07, (New York, NY, USA), p. 234–243, Association for Computing Machinery, 2007.

[3] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Inf. Softw. Technol.*, vol. 52, p. 972–990, sep 2010.

[4] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "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, pp. 268–279, IEEE, 2015.

[5] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing ir-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 324–336, Association for Computing Machinery, 2020.

[6] A. S. Foundation, "Apache lucene core," 2023.

[7] CQSE-GmbH, "Teamscale," 2023.

[8] R. Just, "Defects4j: A database of reproducible bugs in java projects." https://github.com/rjust/defects4j, Year. Accessed on Date.