Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

An Approach for Traceability Recovery between Bug Reports and Test Cases

Guilherme Monteiro Gadelha

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Computer Science

Linha de Pesquisa: Software Engineering

Franklin Ramalho and Tiago Massoni

(Supervisors)

Campina Grande, Paraíba, Brasil

G124a Gadelha, Guilherme Monteiro.

An approach for traceability recovery between bug reports and test cases / Guilherme Monteiro Gadelha. – Campina Grande, 2019.

119 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.

"Orientação: Prof. Dr. Franklin de Souza Ramalho, Prof. Dr. Tiago Lima Massoni".

Referências.

1. Software Engineering. 2. Bug Reports. 3. System Features. 4. Test Cases. 5. Requirements Traceability. 6. Information Retrieval. I. Ramalho, Franklin de Souza. II. Massoni, Tiago Lima. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

“AN APPROACH FOR TRACEABILITY RECOVERY BETWEEN BUG REPORTS AND TEST CASES”

GUILHERME MONTEIRO GADELHA

DISSERTAÇÃO APROVADA EM 23/07/2019

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador(a)

TIAGO LIMA MASSONI, Dr., UFCG
Orientador(a)

EVERTON LEANDRO GALDINO ALVES, Dr., UFCG
Examinador(a)

NATASHA CORREIA QUEIROZ LINO, Dra., UFPB
Examinador(a)

CAMPINA GRANDE – PB

Resumo

Recuperação de links de rastreabilidade automaticamente entre artefatos de software potencialmente melhora o processo de desenvolvimento de software, ajudando a detectar problemas mais cedo durante o ciclo de vida do software. Abordagens que aplicam técnicas de Recuparação da Informação ou Aprendizam de Máquina em dados textuais têm sido propostas, contudo estas técnicas diferem consideravelmente em termos de parâmetros de entrada e resultados obtidos.

É difícil distinguir os benefícios e as falhas das técnicas quando essas são aplicadas isoladamente, usualmente em projetos pequenos ou de tamanho médio. Além disso, um visão mais abrangente poderia ser feita se uma técnica de Aprendizagem Profunda fosse aplicada em comparação com as técnicas tradicionais de Recuperação da Informação.

Nós propomos uma abordagem para recuperar links de rastreabilidade entre artefatos textuais de software, especificamente relatórios de falhas e casos de teste, que são relacionados através de técnicas de Recuperação da Informação e Aprendizagem Profunda. Para avaliar a efetividade de cada técnica, nós usamos um conjunto de dados históricos do Mozilla Firefox usados pelos times de controle de qualidade.

As seguintes técnicas de Recuperação da Informação foram estudadas: Latent Semantic Indexing, Latent Dirichlet Allocation e Best Match 25. Adicionalmente, nós também aplicamos uma técnica de Aprendizagem Profunda chamada Word Vector. Uma vez que não possuímos uma matriz de rastreabilidade que ligue diretamente relatórios de falhas e casos de teste, nós usamos system features como artefatos intermediários.

No contexto de rastreabilidade entre relatórios de falhas e casos de teste, nós identificamos uma performance pobre de três entre as quatro técnicas estudadas. Apenas a técnica Latent Semantic Indexing apresenta resultados satisfatórios, mesmo que comparando com a técnica estado-da-arte Best Match 25. Ao passo que a técnica Word Vector apresentou a efetividade mais baixa dentre todas as técnicas.

Os resultados obtidos mostram que a aplicação da técnica Latent Semantic Indexing – em conjunto com uma combinação de limiares que definem se um link candidato é positivo ou não – é viável para projetos grandes e reais usando um processo de recuperação de

links de rastreabilidade semi-automático, onde os analistas humanos são auxiliados por uma ferramenta de software apropriada.

Abstract

Automatic traceability recovery between software artifacts potentially improves the process of developing software, helping detect issues early during its life-cycle. Approaches applying Information Retrieval (IR) or Machine Learning (ML) techniques in textual data have been proposed, but those techniques differ considerably in terms of input parameters and results.

It is difficult to assess their benefits and drawbacks when those techniques are applied in isolation, usually in small and medium-sized software projects. Also, an overview would be more comprehensive if a promising Deep Learning (DL) based technique is applied, in comparison with traditional IR techniques.

We propose an approach to recover traceability links between textual software artifacts, in special bug reports and test cases, which can be instantiated with a set of IR and DL techniques. For applying and evaluating our solution, we used historical data from the Mozilla Firefox quality assurance (QA) team, for which we assessed the following IR techniques: Latent Semantic Index (LSI), Latent Dirichlet Allocation (LDA) and Best Match 25 (BM25). We also applied the approach with a DL technique called Word Vector. Since there are no traces matrices that straightly link bug reports and test cases, we used system features as intermediate artifacts.

In the context of traceability from bug reports to test cases, we noticed poor performances from three out of the four studied techniques. Only the LSI technique presented satisfactory effectiveness, even standing out over the state-of-the-art BM25 technique. Whereas the Word Vector technique presented the lowest effectiveness in our study.

The obtained results show that the application of the LSI technique – set up with an appropriate combination of thresholds to define if a candidate trace is positive or not – is feasible for real-world and large software projects using a semi-automatized traceability recovery process, where the human analysts are aided by an appropriated software tool.

Agradecimentos

Aos meus dedicados pais, **Kalina Gadelha** e **Hermano Gadelha**, por todo amor, suporte e dedicação durante todos os momentos da minha vida. Ao meu irmão, **Henrique Gadelha**, pelo companherismo e amizade durante toda a vida.

À minha namorada, **Thássia Borges**, pelos incentivos, apoio e carinho nessa jornada. Aos meus sogros, pelos conselhos e acolhimento. Às minhas tias e tios, pelo exemplo de corretude e caráter. Ao meu avô e avós, pelo exemplo de vida.

Aos meus orientadores, **Franklin Ramalho** e **Tiago Massoni**, pela confiança, incentivos e ajuda durante o mestrado. Assim como pela dedicada orientação e solicitude sempre em prol de um trabalho de qualidade.

Aos meus colegas, amigos e professores do SPLab e da UFCG, sem os quais esse trabalho não seria possível. Um agradecimento especial ao professor e amigo **Matheus Gaudencio**, pelos conselhos e boas conversas sobre a vida acadêmica. Agradeço também aos meus colegas de mestrado, em especial, **Marcos Nascimento**, **José Raul**, **Jaziel Moreira** e **Rafael Pontes**, pelas muitas ajudas e pelo altruísmo.

À todos os não citados aqui reconheço e agradeço a importância de vocês em cada vitória conquistada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development and testing processes produce many textual artifacts, such as bug reports, test cases, requirements documents, besides source code itself. The produced artifacts do have interrelations whose tracking benefits software teams. This is especially important for *requirements*, from which several artifacts at many levels of abstraction are closely related. Requirements Traceability is "the ability to describe and follow the life of a requirement, in both a forward and backward direction" [18];

The process of recovering traceability links is split in four steps: (i) parsing of the documents being traced, with the extraction of most relevant words; (ii) the extracted words form a *term-by-document* matrix which supports the traceability recovery made by a computational technique; (iii) it recovers the traceability links between two groups of textual artifacts, for example, mapping bug reports to test cases; the computational technique then ranks the most similar target documents based on a query (source document); (iv) in the end, the rankings are compared with an oracle (ground truth) to evaluate the created links.

Scalable traceability requires automation; if manually maintained, it becomes an error-prone and expensive task [23; 13]. Traceability tools and techniques emerged in response to that demand, allowing traceability links between any textual artifacts to be fastly recovered and analyzed, even in highly dynamic and distributed environments. Although, the tool's effectiveness for application in real projects is an open challenge yet. Information Retrieval (IR) techniques are the basis of most of the proposed techniques for traceability recovery. Antoniol *et al.* were pioneers in using IR techniques for traceability between source code and documentation artifacts in a seminal paper [2].

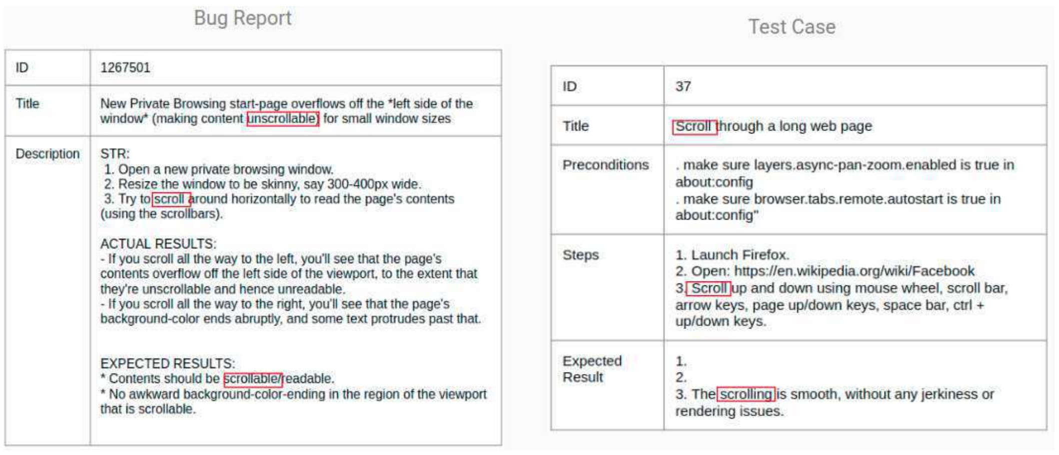
Following the seminal paper, which used the Vector Space Model (VMS) technique, many other works were developed using other techniques, such as Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), and Best Match 25 (BM25). Borg *et al.* [5] verified in a systematic literature review the most common techniques are LSI and VSM, although the BM25 is the state-of-the-art technique in the field.

1.1 Problem

IR techniques differ considerably in terms of input parameters and results. Solutions proposed by previous research [23; 7; 34; 28] for traceability recovering between software artifacts explore a specific technique, such as Latent Semantic Indexing (LSI) or Latent Dirichlet Allocation (LDA), gain from its benefits, but are exposed to its limitations. It is then hard to judge what are the most efficient IR techniques for establishing a useful basis for requirements traceability, which is even harder as the most cited studies focus on small and medium-sized software projects (see discussion regarding related work in Chapter 7). Furthermore, studies using Machine Learning (ML) and Deep Learning (DL) techniques/models for requirements traceability have been carried out, focusing on either requirements identification [12], number of remaining traceability links estimation [15], or traceability links prediction [19]. However, they should be compared with IR techniques for assessing their effectiveness.

An example of link between two artifacts we would like the techniques make automatically is depicted in Figure 1.1. These two artifacts should be linked considering we have a large number of test cases made during the Mozilla Firefox web browser development cycle and a large number of bugs reported. In the left-hand side of Figure 1.1, we observe a bug report and at the right-hand side we observe a test case that we expect a IR/DL technique would be able to recover. The most relevant words the technique can consider are highlighted in red, both artifacts are related with a system feature from the Mozilla Firefox responsible for the smooth *scroll*ing of web pages. The bug report describes problems in scrolling a web page in small-size windows of the Mozilla Firefox, while the manual test case describes how to test the scrolling of a long web page.

Previous studies address traceability recovery between many types of software artifacts,



Bug Report

ID	1267501
Title	New Private Browsing start-page overflows off the *left side of the window* (making content unscrollable) for small window sizes
Description	STR: 1. Open a new private browsing window. 2. Resize the window to be skinny, say 300-400px wide. 3. Try to scroll around horizontally to read the page's contents (using the scrollbars). ACTUAL RESULTS: - If you scroll all the way to the left, you'll see that the page's contents overflow off the left side of the viewport, to the extent that they're unscrollable and hence unreadable. - If you scroll all the way to the right, you'll see that the page's background-color ends abruptly, and some text protrudes past that. EXPECTED RESULTS: * Contents should be scrollable/readable. * No awkward background-color-ending in the region of the viewport that is scrollable.

Test Case

ID	37
Title	Scroll through a long web page
Preconditions	. make sure layers.async-pan-zoom.enabled is true in about:config . make sure browser.tabs.remote.autostart is true in about:config"
Steps	1. Launch Firefox. 2. Open: https://en.wikipedia.org/wiki/Facebook 3. Scroll up and down using mouse wheel, scroll bar, arrow keys, page up/down keys, space bar, ctrl + up/down keys.
Expected Result	1. 2. 3. The scrolling is smooth, without any jerkiness or rendering issues.

Figure 1.1: Bug report and test case that should be linked

but we have noticed only a few studies *tracing bug reports to test cases* [5]. Although test cases very often are the most up-to-date documentation of the system and the only available source of system requirements, especially in agile development teams. One previous related study works with traceability between requirements and bug reports [42]; Hemmati *et al.* [24] investigate IR techniques for predicting manual test case failure; Merten *et al.* [31] analyzed variations of five IR techniques for traceability recovery between bug reports; while only one study deals with traceability between test cases and bug reports [25]. See Chapter 7 for more details on these two last studies.

In summary, to the best of our knowledge, we have no clear indications about the best technique to use for traceability recovery between bug reports and test cases. Also, we did not find studies providing satisfactory results for large and real-world projects to adopt a traceability recovery process between these two kinds of artifacts. So far, the few works found tracing bug reports and test cases have limitations on the variety of techniques studied and on the depth of the evaluation made, many times not estimating the impact of the traceability recovery chosen technique over the software development and maintenance processes. Our study proposes an approach to fill that gap and help to link appropriately these two important software artifacts.

1.1.1 Problem Relevance

An effective traceability recovery process promotes reliable software development and testing, as it has a significant repercussion over activities such as change impact estimation, test selection, and prioritization, and budget prediction [19]. In such a scenario, bugs reported by developers, testers, final users, and stakeholders can be selected and prioritized for bug fixing and testing processes automatically using IR or DL techniques. Also, the impact of changes suggested by stakeholders may be more precisely estimated if information about the affected artifacts is available for decision makers, reducing the involved risks for the project. The same information can be used for budget prediction, once more data are available for estimating the teams' sizes and the number of hours required for bug fixing for example. Yet another contribution of such process is the identification of the reason of the faults, once the correct linking between the bug reports and test cases improve the developers' capabilities of identifying software non-conformances and faults.

Another benefit is the reduction in the learning curve required from new members of development or testing teams. Usually, the projects do not have up-to-date documentation of all software artifacts and new team members must learn in practice the localization of artifacts and the architecture of the software, such learning takes time and effort from all the involved team members. The use of traceability recovery tools may reduce this learning process and speed up the integration of these new members for more critical activities into the project.

1.2 Scope

The scope of our research is limited to the field of Requirements Engineering, specifically the sub-field of Requirements Traceability. Despite the fact "Requirements" be related to a specific kind of software artifact (requirements documents), considering the community's vocabulary, we understand it as any type of software textual artifact.

1.3 Objectives

In this dissertation, we propose an approach (Chapter 3) that applies a set of IR and DL techniques to recover traceability links between bug reports and test cases. How we do not have a traceability matrix which maps directly bug reports and test cases, we used system features as intermediate artifacts. The system features allowed us to group the test cases and helped us in the generation of an oracle – which means the ground thruth –, so we could evaluate each IR and DL technique. The selected techniques are Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), Best Match 25 (BM25), and Word Vector.

For evaluating our solution, we used historical data from the Mozilla Firefox development team[1]. We have designed and set up and performed one empirical study; The study is split in two phases, the first phase (Chapter 4) is to generate a traceability matrix (oracle) between features and bug reports. The second phase (Chapter 5) uses this traceability matrix to generate a traceability matrix between *bug reports and test cases*, and finally evaluates the IR and DL techniques for those artifacts. The main *goal* of this study is to compare available IR and DL techniques in one specific context of traceability recovery: bug reports to test cases, in terms of well-known metrics: $Precision$, $Recall$, and F_2-$Score$.

The *main objective* of this work is the analysis and discussion about the effectiveness of a varied group of IR and DL techniques through the reporting of different metrics should grant the community a deeper understanding of the studied techniques when using them for traceability recovery of the kind of used artifacts in real and open source projects such as the Mozilla Firefox.

The study has the following secondary objectives:

1. Organization of a data set of bug reports, test cases, and system features based on Mozilla Firefox available artifacts for using in other traceability recovery studies and support the creation of new benchmarks of traceability recovery techniques;

2. Creation of an approach for traceability recovery between bug reports and test cases using system features;

[1]https://www.mozilla.org/

1.4 Research Questions

For this research work, we defined the following research questions. We consider all research questions in the context of traceability between bug reports and test cases and more information on the metrics used can be found in Chapter 5:

RQ1 *Which technique presents the best effectiveness?*

Where the technique's *effectiveness* is understood in terms of $Precision$, $Recall$, and F_2-$Score$.

RQ2 *How does the effectiveness of each technique compare with a baseline predictor effectiveness?*

We used as baseline predictor of a ZeroR Classifier/Predictor implementation, which only predicts the *majority class*.

RQ3 *How does the effectiveness of each technique vary based on variable cuts?*

This third research question explores the impact of a variety of determined combinations of Similarity Thresholds and Top Values over the effectiveness of the techniques. Similarity Thresholds and Top Values are defined so the number of documents returned from a query can be limited.

RQ4 *Which technique presents the best Goodness?*

The well-known *Goodness* scale allows us to understand the feasibility of a technique for application into a traceability recovery process.

RQ5 *Which technique presents the lowest REI coefficient?*

This last research question should allow us to compare the techniques between themselves in terms of effort saving in a traceability recovery process from a human analyst. The *REI* abbreviation stands for Recovery Effort Index.

1.5 Dissertation Structure

This dissertation is organized as follows: Chapter 2 exposes basic concepts needed to understand our proposal; Chapter 3 details and schematizes our proposal; whereas Chapter 4 presents the first empirical study, relative to the bug reports to system features oracle creation, and discusses its results. Continuing the analysis, Chapter 5 discusses the bug reports to test cases oracle creation, and evaluates the application of the IR and DL techniques. The threats to the validity of the conclusions of the study are reported by Chapter 6. The Chapter 7 presents some related works, discussing similarities and differences with our study; and, for last, Chapter 8 exposes our conclusions and future works we intend to develop. Appendix A presents an additional study on traceability between bug reports and features similar to the one exposed at Chapter 5.

Chapter 2

Background

In this chapter, we introduce the concepts related to bug reports, test cases and system features, which are the software artifacts used in our approach. Also, we define the IR and DL techniques selected for our study.

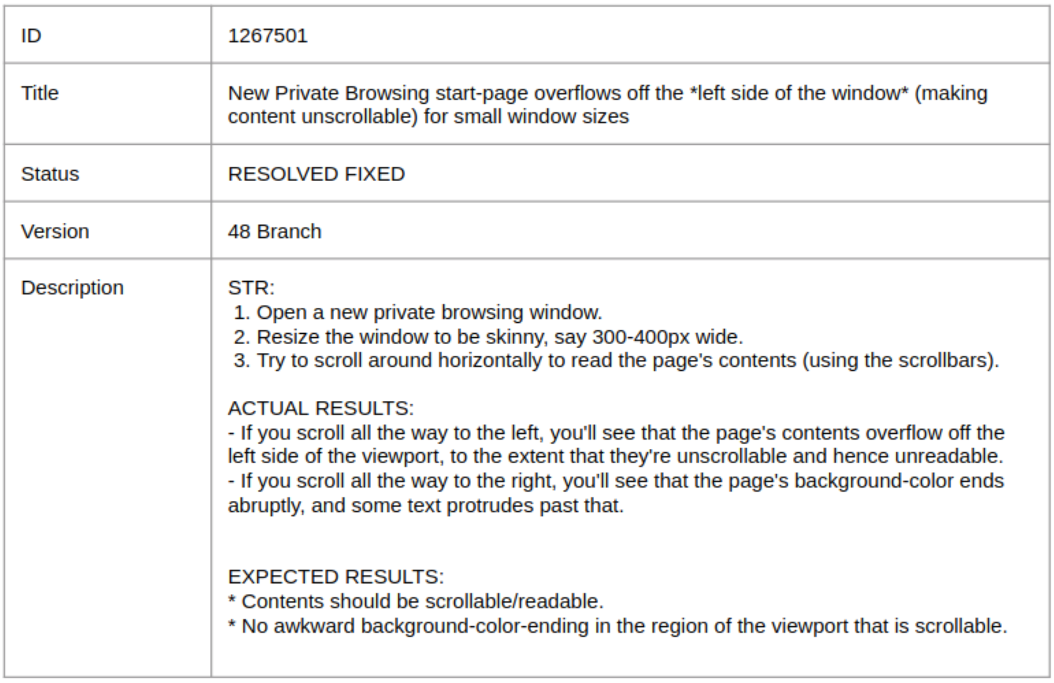
2.1 Bug Reports

A bug report describes any system failure identified by a user or automatically reported by the system (in the case of crashing bugs) [16]. The main objective of a bug report is to offer details about a failure identified and then help the developers investigate the causes and fix the issue if its presence is confirmed [27]. Bugs occur due to either implementation faults or specification nonconformances that will be detected by end users during the system's operation. Several fields may be added to the reports, including title, reproduction steps, stack traces, failing test cases, expected behavior, among other data [8]. Figure 2.1 shows an example of a bug report from the Mozilla Firefox repository[1].

A bug report defined appropriately is decisive for debugging and bug fixing. It should contain a clear and detailed problem description; the procedure taken to reproduce the bug has to be accurate and include precise information about inputs and outputs. Complete information about observed and expected behavior is associated with bug acceptance by developers and its successful resolution [43].

In Figure 2.1, we identify attributes which qualify a bug report and contribute to its

[1]https://bugzilla.mozilla.org



ID	1267501
Title	New Private Browsing start-page overflows off the *left side of the window* (making content unscrollable) for small window sizes
Status	RESOLVED FIXED
Version	48 Branch
Description	STR: 1. Open a new private browsing window. 2. Resize the window to be skinny, say 300-400px wide. 3. Try to scroll around horizontally to read the page's contents (using the scrollbars). ACTUAL RESULTS: - If you scroll all the way to the left, you'll see that the page's contents overflow off the left side of the viewport, to the extent that they're unscrollable and hence unreadable. - If you scroll all the way to the right, you'll see that the page's background-color ends abruptly, and some text protrudes past that. EXPECTED RESULTS: * Contents should be scrollable/readable. * No awkward background-color-ending in the region of the viewport that is scrollable.

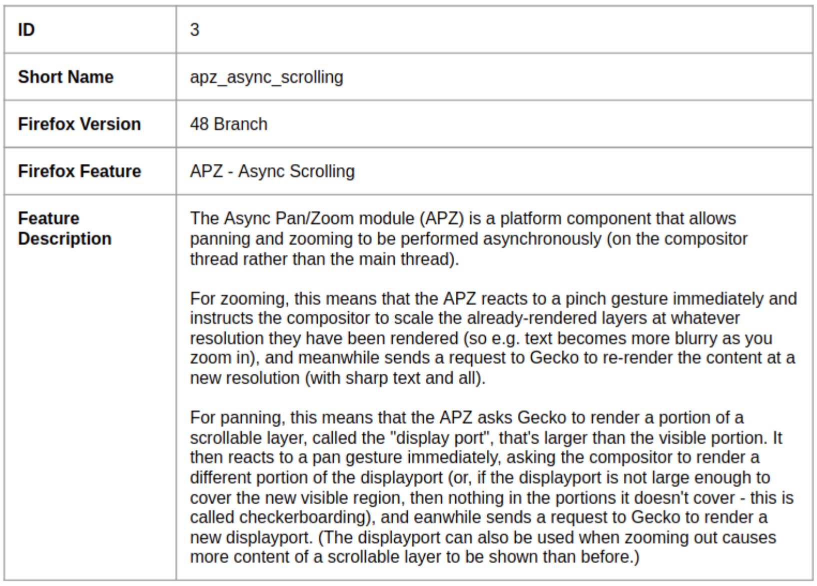
Figure 2.1: Example of Bug Report from Mozilla Firefox

acceptance by the Mozilla's development team: (i) it has a unique ID number; (ii) the steps to reproduce – STR – are clearly described; (iii) the expected results are detailed; and (iv) the problem is summarized and very specific, as we observe in the Title field.

2.2 System Features

A system feature is defined as a set of requirements highly bonded to each other [26]. System features improve the communication efficacy between all the stakeholders by virtue of the common vocabulary created and the simpler cognitive effort needed for understanding them in comparison to individual requirements [35]; requirements itself are well understood by those that define them and those which implement them. The definition of system features creates a common ground so that every person involved in the project can quickly understand the system operations.

A feature is commonly described by its *Name* and *Description*, but others fields such as *Software Version*, which favors the features traceability, can also be used depending on the model of representation adopted by the system's managers. An example of system feature



ID	3
Short Name	apz_async_scrolling
Firefox Version	48 Branch
Firefox Feature	APZ - Async Scrolling
Feature Description	The Async Pan/Zoom module (APZ) is a platform component that allows panning and zooming to be performed asynchronously (on the compositor thread rather than the main thread). For zooming, this means that the APZ reacts to a pinch gesture immediately and instructs the compositor to scale the already-rendered layers at whatever resolution they have been rendered (so e.g. text becomes more blurry as you zoom in), and meanwhile sends a request to Gecko to re-render the content at a new resolution (with sharp text and all). For panning, this means that the APZ asks Gecko to render a portion of a scrollable layer, called the "display port", that's larger than the visible portion. It then reacts to a pan gesture immediately, asking the compositor to render a different portion of the displayport (or, if the displayport is not large enough to cover the new visible region, then nothing in the portions it doesn't cover - this is called checkerboarding), and eanwhile sends a request to Gecko to render a new displayport. (The displayport can also be used when zooming out causes more content of a scrollable layer to be shown than before.)

Figure 2.2: Example of System Feature from Mozilla Firefox

can be observed in Figure 2.2[2].

The Figure 2.2 shows the APZ – *Async Pan/Zoom* – system feature from Mozilla Firefox. This feature is related to the bug report exemplified in the previous section. The APZ feature is responsible for the performance improvement in the panning and zooming actions within the Firefox browser, separated from the main javascript thread.

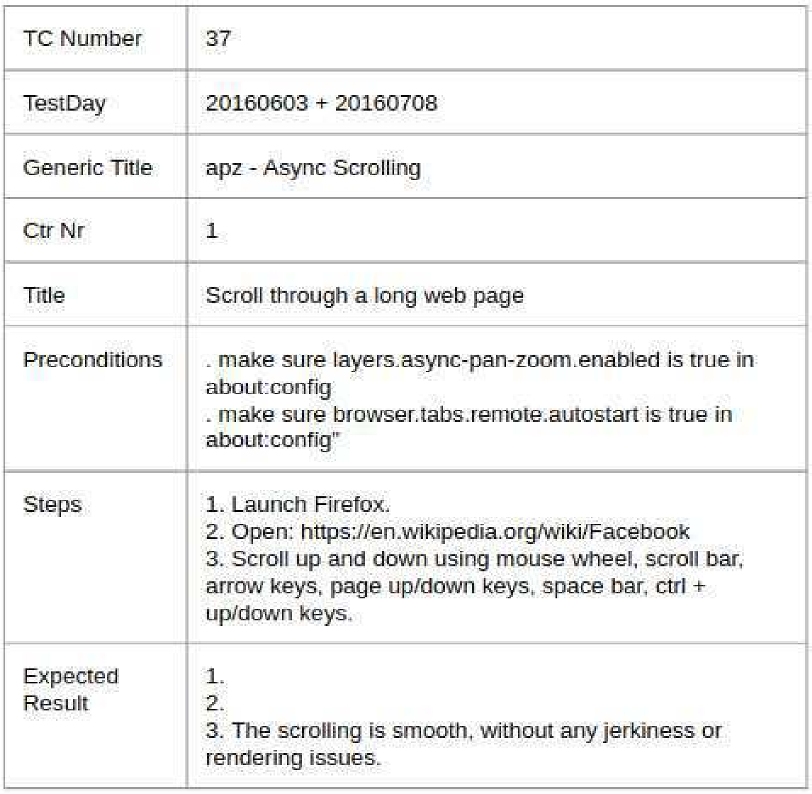
The Firefox features are organized following the presented template in the Figure 2.2, but every project has a particular way of organizing the documentation of their features, when this is made. In our proposed template, we characterize a system feature using the following arguments: *ID*, *Short Name*, *Firefox Version*, *Firefox Feature* (Feature Name), and *Feature Description*.

2.3 Test Cases

A manual test case defines a textual sequence of steps, each of which including the expected results, allowing a tester to determine if a software is following the stakeholder's requirements [39]. Figure 2.3 shows an example of test case from the Mozilla Firefox, which is related with the previously presented Bug Report (Section 2.1) and System Feature (Sec-

[2]https://wiki.mozilla.org/Platform/GFX/APZ

tion 2.2). Besides the test case *Title*, *Steps to Reproduce* and *Expected Results*, the related *TestDay*, *TC Number*, *Generic Title*, *Preconditions*, and *Crt Nr* (Control Number?) are also detailed.



TC Number	37
TestDay	20160603 + 20160708
Generic Title	apz - Async Scrolling
Ctr Nr	1
Title	Scroll through a long web page
Preconditions	. make sure layers.async-pan-zoom.enabled is true in about:config . make sure browser.tabs.remote.autostart is true in about:config"
Steps	1. Launch Firefox. 2. Open: https://en.wikipedia.org/wiki/Facebook 3. Scroll up and down using mouse wheel, scroll bar, arrow keys, page up/down keys, space bar, ctrl + up/down keys.
Expected Result	1. 2. 3. The scrolling is smooth, without any jerkiness or rendering issues.

Figure 2.3: Example of Test Case from Mozilla Firefox

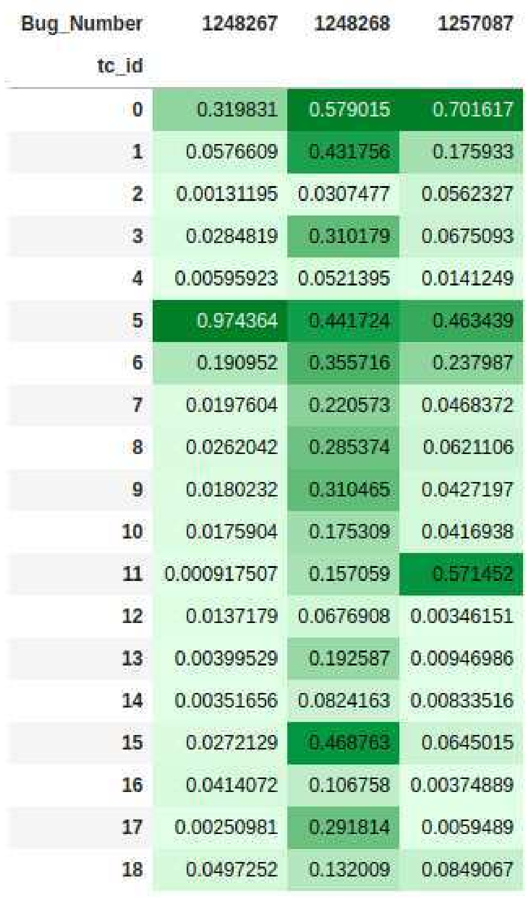
The test case's *Title* is a short description of the test purpose, which should be executed after the *Preconditions* be attended and following the *Steps to Reproduce*, then for each step an *Expected Result* is defined and the agreement with it must be checked by the tester. If the expected results match with the program outputs, then the test passes, otherwise, it fails.

The *TC Number* is a unique ID for the test case, so it can be identified between all test cases. Especially in the case of Mozilla Firefox, a test case is always associated with a system feature (*Generic Title*) and with at least one *TestDay*, which is the day the test was executed. We have not identified the semantic of the *Crt Nr* field, we estimate it is a unique identifier for the test in the *TestDay*.

Test Cases have fundamental importance for the software development, evolution, and maintenance, once allow the detection of bugs before the software be released to the final users or ahead of prejudices for any stakeholder. Also, the test cases are easy to automate and facilitate the tests and bugs reproducibility. Besides that, test cases are the most up-to-date documentation of many systems in the industry.

2.4 IR and DL Techniques for Traceability Recovery

The purpose of Information Retrieval (IR) [6] techniques is to recover and rank a set of documents from a corpus of documents when a query is submitted to it. The output of an applied IR technique in the context of traceability recovery is a *similarity matrix* [2] which does the mapping between each pair (document, query). The content of the *similarity matrix* is the similarity scores between the documents and the queries, and such scores are calculated in many different ways accordingly to the techniques core algorithms.



Bug_Number	1248267	1248268	1257087
tc_id			
0	0.319831	0.579015	0.701617
1	0.0576609	0.431756	0.175933
2	0.00131195	0.0307477	0.0562327
3	0.0284819	0.310179	0.0675093
4	0.00595923	0.0521395	0.0141249
5	0.974364	0.441724	0.463439
6	0.190952	0.355716	0.237987
7	0.0197604	0.220573	0.0468372
8	0.0262042	0.285374	0.0621106
9	0.0180232	0.310465	0.0427197
10	0.0175904	0.175309	0.0416938
11	0.000917507	0.157059	0.571452
12	0.0137179	0.0676908	0.00346151
13	0.00399529	0.192587	0.00946986
14	0.00351656	0.0824163	0.00833516
15	0.0272129	0.468763	0.0645015
16	0.0414072	0.106758	0.00374889
17	0.00250981	0.291814	0.0059489
18	0.0497252	0.132009	0.0849067

Figure 2.4: Similarity Matrix Example

In our work, the bug reports are used as queries, whereas the test cases are used as the documents. An example of a similarity matrix between bug reports and test cases is shown in Figure 2.4. The dark green cells show higher levels of similarity. For example, the similarity score between the bug report 1248268 and the test case 5 is 0.4417. In general is used the cosine function to calculate the similarity – as we later explain –, however other functions could be also used, such as Edit Distance, Jaccard Similarity, Dice Similarity, etc. [5]. We decided to use the cosine following previous works methodologies [22; 23; 9; 10].

In order to improve the IR algorithms performance, before the submission to the technique's core algorithms – later explained in this Chapter –, the corpus documents and the set of queries pass through a preprocessing phase. The first step is the process of tokenization of each document, where blank spaces and punctuation characters are removed. In the second step, the set of tokens is submitted to the removal of stop words to discard terms like articles, adverbs, and prepositions. In the third step is performed a morphological analysis applying stemming in each token which removes words suffixes, so words like ***informat***ion, ***informat***ics, and ***informat***ization are treated as one; and lemmatization, where words in the third person are changed to first person and verbs in past and future tenses are changed into the present tense.

Deep Learning techniques also can be used for calculating the similarity between a given pair of documents. Based on recent works [12; 19], we also used a Deep Learning technique as a traceability recovery technique. The technique details are explained in Section 2.4.4.

In the sequel, we detail each technique used in this work. We chose them based on the analysis of a systematic literature review [5] and previous works [4; 34; 38; 12; 19].

2.4.1 Latent Semantic Indexing

Latent Semantic Indexing (LSI) [11] is a very common IR technique based on a vector space model [5; 9; 13]. The technique requires the vectorization of each document in the corpus and each query in the queries set. In order to do this vectorization, a weighting scheme is selected so the most relevant words of each document and query can have the appropriate weight in the searching and ranking process. One of the most common weighting schemes used in the LSI is called *tf-idf* which stands for *term frequency-inverse document frequency*. The *tf-idf* formula is detailed in Equation 2.1.

$$tf\text{-}idf(t,d,D) = tf(t,d).idf(t,D) \tag{2.1}$$

The first part of the formula – *tf(t,d)* – calculates the frequency of the term *t* in a document *d*, so how much more the term appears in the document, higher the *tf* value. The second part calculates the number of documents the term *t* appears in the entire corpus of documents *D*, so how much more rare the term is, the higher the *idf(t,D)* value.

$$idf(t,D) = \ln(\frac{N}{1+n_t}) \tag{2.2}$$

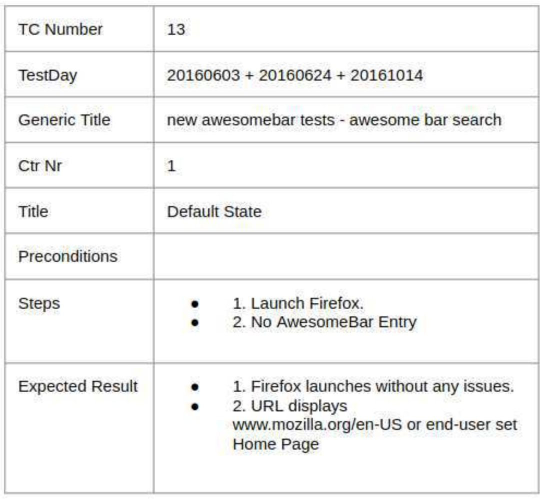
The Equation 2.2 details the smoothed *idf* formula, where N is the number of documents in the corpus, and n_t is the number of documents where the term t appears. How the n_t value can be zero, resulting in a division by zero, the equation is smoothed by summing 1 in the denominator.

Using the matrix of *term-by-document* whose content is $tf\text{-}idf$ values as input, a mathematical dimensionality reduction technique known as SVD (Singular Value Decomposition) [11] is applied and we have as output new vectors representing documents and queries. This mathematical technique is needed to optimize the LSI's effectiveness and to make the searching process faster. The similarity score between each pair (document, query) is calculated by the cosine of the angle between a document vector and a query vector.

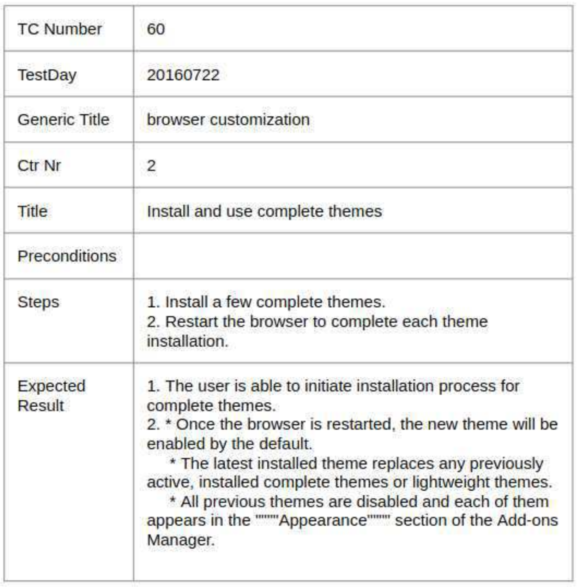
In order to improve the understanding of the LSI technique, we present a concrete example with one bug report and three test cases. The bug report used is the same presented in Section 2.1, such as the test case which was presented in Section 2.3. Two other test cases used in our example are displayed in Figure 2.5.

We refer to the bug report and test cases used through their unique identifiers, so the bug report becomes *BR_1267501_SRC* and the test cases become *TC_13_TRG*, *TC_37_TRG*, and *TC_60_TRG*. The abbreviations SRC and TRG stand for source and target, meaning the direction of the traceability recovery, from the bug report (source) to the test cases (targets).

We started the recovering process by executing the preprocessing phase described at the beginning of the section, applying tokenization, stemming, stop words removal, etc. in the test cases and in the bug report. Also, the tokens were sorted alphabetically and converted to lowercase. The LSI's application for traceability recovering is presented in Figure 2.6.

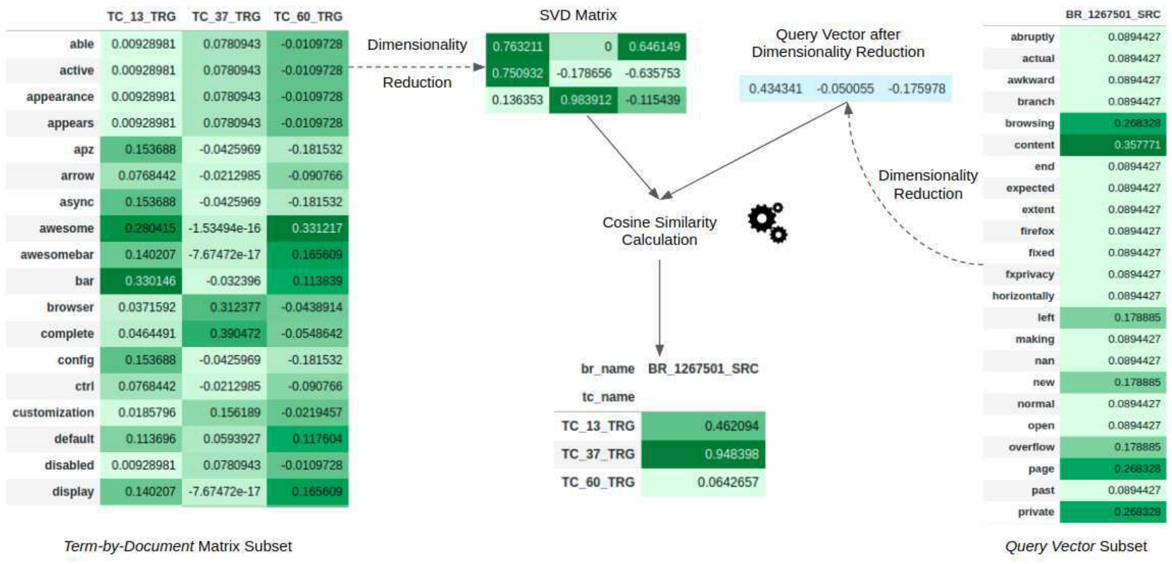


TC Number	13
TestDay	20160603 + 20160624 + 20161014
Generic Title	new awesomebar tests - awesome bar search
Ctr Nr	1
Title	Default State
Preconditions	
Steps	• 1. Launch Firefox. • 2. No AwesomeBar Entry
Expected Result	• 1. Firefox launches without any issues. • 2. URL displays www.mozilla.org/en-US or end-user set Home Page



TC Number	60
TestDay	20160722
Generic Title	browser customization
Ctr Nr	2
Title	Install and use complete themes
Preconditions	
Steps	1. Install a few complete themes. 2. Restart the browser to complete each theme installation.
Expected Result	1. The user is able to initiate installation process for complete themes. 2. * Once the browser is restarted, the new theme will be enabled by the default. * The latest installed theme replaces any previously active, installed complete themes or lightweight themes. * All previous themes are disabled and each of them appears in the """"Appearance"""" section of the Add-ons Manager.

Figure 2.5: Test Cases 13 and 60 used in our example



	TC_13_TRG	TC_37_TRG	TC_60_TRG
able	0.00928981	0.0780943	-0.0109728
active	0.00928981	0.0780943	-0.0109728
appearance	0.00928981	0.0780943	-0.0109728
appears	0.00928981	0.0780943	-0.0109728
apz	0.153688	-0.0425969	-0.181532
arrow	0.0768442	-0.0212985	-0.090766
async	0.153688	-0.0425969	-0.181532
awesome	0.280415	-1.53494e-16	0.331217
awesomebar	0.140207	-7.67472e-17	0.165609
bar	0.330146	-0.032396	0.113839
browser	0.0371592	0.312377	-0.0438914
complete	0.0464491	0.390472	-0.0548642
config	0.153688	-0.0425969	-0.181532
ctrl	0.0768442	-0.0212985	-0.090766
customization	0.0185796	0.156189	-0.0219457
default	0.113696	0.0593927	0.117604
disabled	0.00928981	0.0780943	-0.0109728
display	0.140207	-7.67472e-17	0.165609

Term-by-Document Matrix Subset

SVD Matrix

0.763211	0	0.646149
0.750932	-0.178656	-0.635753
0.136353	0.983912	-0.115439

Query Vector after Dimensionality Reduction

0.434341	-0.050055	-0.175978

br_name	BR_1267501_SRC
tc_name	
TC_13_TRG	0.462094
TC_37_TRG	0.948398
TC_60_TRG	0.0642657

	BR_1267501_SRC
abruptly	0.0894427
actual	0.0894427
awkward	0.0894427
branch	0.0894427
browsing	0.268328
content	0.357771
end	0.0894427
expected	0.0894427
extent	0.0894427
firefox	0.0894427
fixed	0.0894427
fxprivacy	0.0894427
horizontally	0.0894427
left	0.178885
making	0.0894427
nan	0.0894427
new	0.178885
normal	0.0894427
open	0.0894427
overflow	0.178885
page	0.268328
past	0.0894427
private	0.268328

Query Vector Subset

Figure 2.6: LSI Example

The matrix depicted on the left side of Figure 2.6 is called *Term-by-Document Matrix* and is created with the terms presented in the test cases. The most important terms in each document are highlighted in dark green, the weights of the terms are calculated using the tf-idf scheme. Similarly, the query vector – created based on the existing terms in the *Term-by-Document Matrix* and also presented in the bug report – is depicted in the right side of Figure 2.6.

The $tf\text{-}idf$ weights are calculated such as in the following example. Consider the bug report presented in Section 2.1 containing 200 words wherein the word "apz" appears 5 times. The term frequency (*i.e.*, tf) for "apz" is then $\frac{5}{200} = 0.025$. Now, assume we have 300 test cases and the word "apz" appears in 35 of these. Then, the inverse document frequency (*i.e.*, idf) is calculated as $\ln(\frac{300}{35}) = 8.57$. Thus, the $tf\text{-}idf$ weight is the product of these quantities: $0.025 * 8.57 = 0.2142$ [1].

Following the *Term-by-Document Matrix* creation, the SVD mathematical technique is applied over it, generating a *SVD Matrix* of smaller dimensions (3x3), and over the *Query Vector*, generating a reduced vector with dimensions (1x3). Then, the cosine similarity is calculated between each test case (line in the SVD matrix) and the bug report (reduced query vector), resulting in the vector depicted at the center in the Figure 2.6.

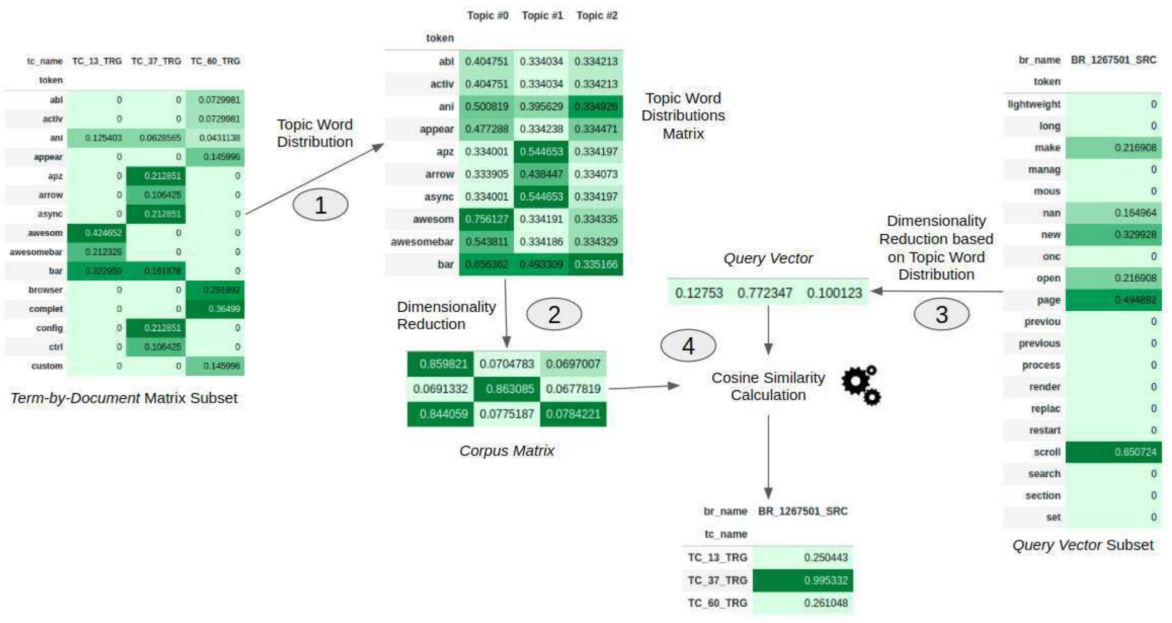
We see, after the Cosine Similarity Calculation, the LSI was able to correctly recover the test case 37 from the analysis of the bug report 1267501 with a high similarity score of 0.9483 (dark green), and a low similarity scores with the other two test cases, which are in fact not related with bug report (lighter green).

2.4.2 Latent Dirichlet Allocation

The LDA technique [4] is a generative statistical model [3] where each collection of discrete data as a textual document is modeled as a set of topics and each topic, in turn, is modeled as a mixture of probabilities. Once the topic probabilities provide a representation of a document, the LDA is called a topic model. The referred probabilities are relative to the following question: *What is the probability of a query **q** be entered to retrieve a document **d**?*. A topic model estimates which topics – created based on the content of the documents – are the most representative for a given document and attributes a specific distribution of topics for each one of them. The same way, the topics of a given query are calculated, and then the similarity scores between the query and the documents can be estimated.

Many different metrics can be used to calculate the similarity scores. One of them is to use the cosine of the angle between the vectors of each pair (document, query) as is made in the LSI technique. The difference is that the vectors here are vectors of probabilities [13]. A

[3]Given an observable variable X and a target variable Y, a generative model is a statistical model of the joint probability distribution on $X \times Y$, P(X, Y) [41]



tc_name	TC_13_TRG	TC_37_TRG	TC_60_TRG
token			
abl	0	0	0.0729981
activ	0	0	0.0729981
ani	0.125403	0.0628565	0.0431138
appear	0	0	0.145996
apz	0	0.212851	0
arrow	0	0.106425	0
async	0	0.212851	0
awesom	0.424652	0	0
awesomebar	0.212326	0	0
bar	0.322959	0.161878	0
browser	0	0	0.291992
complet	0	0	0.36499
config	0	0.212851	0
ctrl	0	0.106425	0
custom	0	0	0.145996

	Topic #0	Topic #1	Topic #2
token			
abl	0.404751	0.334034	0.334213
activ	0.404751	0.334034	0.334213
ani	0.500819	0.395629	0.334926
appear	0.477288	0.334238	0.334471
apz	0.334001	0.544653	0.334197
arrow	0.333905	0.438447	0.334073
async	0.334001	0.544653	0.334197
awesom	0.756127	0.334191	0.334335
awesomebar	0.543811	0.334186	0.334329
bar	0.656362	0.493309	0.335166

0.859821	0.0704783	0.0697007
0.0691332	0.863085	0.0677819
0.844059	0.0775187	0.0784221

0.12753	0.772347	0.100123

br_name	BR_1267501_SRC
token	
lightweight	0
long	0
make	0.216908
manag	0
mous	0
nan	0.164964
new	0.329928
onc	0
open	0.216908
page	0.494892
previou	0
previous	0
process	0
render	0
replac	0
restart	0
scroll	0.650724
search	0
section	0
set	0

br_name	BR_1267501_SRC
tc_name	
TC_13_TRG	0.250443
TC_37_TRG	0.995332
TC_60_TRG	0.261048

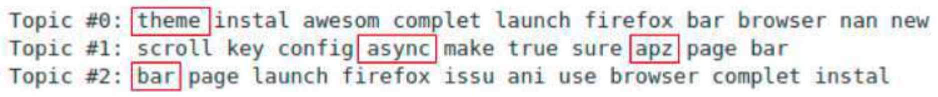
Figure 2.7: LDA Example

concrete example with the application of the LDA technique using the same test cases and bug report cited in the LSI explanation is shown in Figure 2.7.

In the left side of figure, we show the tokens of each test case after $tf\text{-}idf$ application, the result is a *Term-by-Document Matrix* (subset depicted in Figure 2.7). Next, the LDA's topic word distributions are created (step 1), in our case we set up the LDA to have three topics – other values can be chosen, which impacts the technique's effectiveness –, considering the technique will be capable of distinguishing the test cases origins – each test case is related to three different system features from Mozilla Firefox. Observe we are mapping bug reports to test cases, although the technique is able to recognize the different system features associated with the test cases.

In fact, we recognize, through the analysis of the topics distributions, the technique successfully identifies the test cases associated system features: the first topic (*Topic #0)* is referring to the browser customization feature, whereas the second topic (*Topic #1)* is pointing out to scrolling characteristic of APZ system feature, and the third topic (*Topic #2)* is related with the *New Awesome Bar* feature. The ten most relevant words for each topic are detailed in Figure 2.8 and important words are highlighted in red.

After the topic word distribution calculation, a dimensionality reduction operation is applied and a *Corpus Matrix* with dimensions 3x3 is generated (step 2). Identically, a di-



```
Topic #0: theme instal awesom complet launch firefox bar browser nan new
Topic #1: scroll key config async make true sure apz page bar
Topic #2: bar page launch firefox issu ani use browser complet instal
```

Figure 2.8: LDA Topics

mensionality reduction operation is applied over the bug report (query) vector, generating a reduced query vector with dimensions 3x1 (step 3). Then, the cosine similarity is determined for each line of the *Corpus Matrix* (test case) and the reduced query vector (bug report) (step 4). We observe the technique was able to correctly recover the related test case (37), with a high similarity score 0.9953 (darker green cell), while attributed a low similarity score to the other test cases (lighter green cells). Observe the Figure 2.7 shows only subsets of the *Term-by-Document Matrix*, the *Topic Word Distribution Matrix*, and the *Query Vector*.

2.4.3 Best Match 25

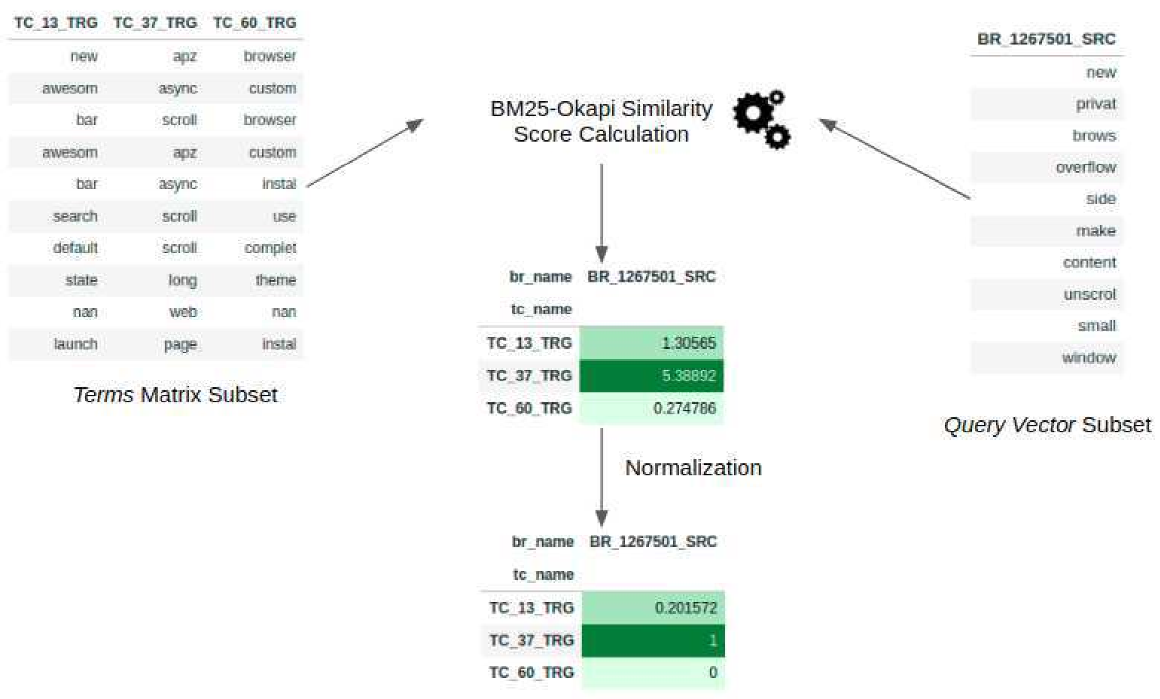
Also known as BM25, the Best Match 25 is a probabilistic model which is based on the Okapi-BM25 scoring function for ranking the retrieved documents [38]. Probabilistic models in the context of information retrieval try to answer the question *What is the probability of a given document be relevant to a given query?* to answer this question, scoring functions are used to give a score to each document and rank the entire set of documents concerning each query. The scoring function of the BM25 model can be generally described by the Equations 2.3 and 2.4 [7].

$$S_d(q) = \sum_{t \in q} W(t) \tag{2.3}$$

$$W(t) = \frac{TF_t(k_1 + 1)}{k_1((1-b) + b.\frac{DL}{AVGDL})} log(\frac{N}{ND_t}) \tag{2.4}$$

The final score of a document *d* in relation to a query *q* is calculated by Equation 2.3, where *t* is each term in the query *q* and *W(t)* is the weight of a specific term *t* for the document *d*. In the Equation 2.4, TF_t is the term frequency in the document d, DL is the document length, $AVGDL$ is the average document length, N is the corpus size, and ND_t is the amount of documents in the corpus that have the term t. The variables k_1 and b are tunneable parameters, where the first one calibrates the effect of term frequency, and second one calibrates the effect of document length.

Similarly we did for the LSI and LDA techniques, we illustrate the BM25 technique application with an example, with the same test cases and bug report. Figure 2.9 shows the technique application. Higher values of similarity are depicted in dark green.



TC_13_TRG	TC_37_TRG	TC_60_TRG
new	apz	browser
awesom	async	custom
bar	scroll	browser
awesom	apz	custom
bar	async	instal
search	scroll	use
default	scroll	complet
state	long	theme
nan	web	nan
launch	page	instal

Terms Matrix Subset

BR_1267501_SRC
new
privat
brows
overflow
side
make
content
unscrol
small
window

Query Vector Subset

br_name	BR_1267501_SRC
tc_name	
TC_13_TRG	1.30565
TC_37_TRG	5.38892
TC_60_TRG	0.274786

br_name	BR_1267501_SRC
tc_name	
TC_13_TRG	0.201572
TC_37_TRG	1
TC_60_TRG	0

Figure 2.9: BM25 Example

For each document in the corpus (set of test cases) we apply the preprocessing phase and extract its tokens, as we see in the *Terms Matrix Subset* at the left side of the figure. The same process is applied in the query (bug report), whose result we observe on the right side of the figure. Then, the BM25-Okapi similarity score is calculated for each combination of a test case and bug report, resulting in a vector of similarity scores greater than zero. In order to compare the BM25's similarity scores with other techniques, we apply the normalization of the obtained scores for the scale [0,1], so the smallest score becomes 0, the higher becomes 1 and other values are calculated in relation to these two reference values in the scale [0,1].

Note the BM25 technique was able to recover the correct trace of the bug report with the test case 37, at the same time it attributed low values of similarity for the other test cases, which in fact are not related with the referred bug report.

2.4.4 Word Vector

Deep Learning (DL) is a family of methods among the Machine Learning methods based on Artificial Neural Networks [17]. These networks characterize themselves for having a high number of hidden layers and being very deep so that they are able of capturing many different patterns present in images data sets, text corpora, and audio records data sets for example when presented to large amounts of these data. Once the deep neural network is trained, it is able to recognize objects in images, translate texts between languages, and do speech recognition between many other applications.

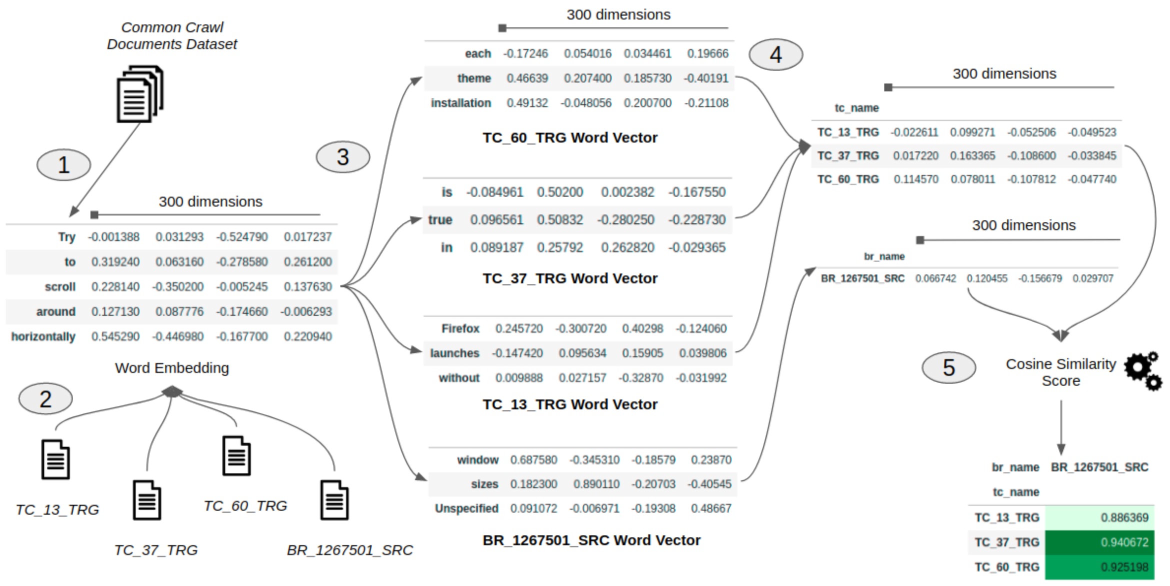
Word Vector is a Deep Learning [17] technique used in this work for traceability recovery, inspired by recently developed studies in the field [12; 19]. The use of word embeddings has become successful with the advancements of Deep Learning, combined with the availability of large amounts of data for training models and increasing computing capabilities to give support to these advancements. The releasing and dissemination of open-source libraries such as Google's word2vec[4] [32] facilitated their use as pre-trained models – trained on very large available data sets – or even the training of new word embeddings.

The word2vec library receives as input a large amount of text, such as the Common Crawl Dataset[5], which is a corpus of collected news, comments, and blogs in the web, and produces as output a vector space model, commonly with hundreds of dimensions. Each unique word (token) is represented by a vector in this space with the same amount of dimensions, and each dimension of this vector is learned during the training of a Convolutional Neural Network (CNN) [17] or another kind of Deep Neural Network. In the context of traceability recovery, the trained neural network is available at the end of the process and can be retrained with the source and target artifacts, so nuances from the domain of these textual documents can be captured by the model and appropriately represented in the vector space model. For example, the context of the word "bug" used to appear in software engineering texts is different from the used in biology texts, and this impacts the representation of the word into the word embedding. Word Embeddings can capture the syntactic *and* semantic relations between words in the text, differently from the previously presented IR techniques. Therefore, the trained model is capable of making some semantic inferences. For example, presenting the

[4]https://github.com/svn2github/word2vec

[5]https://commoncrawl.org/

relationship *(Man,Woman)* for the model, and asking to the corresponding relationship for the word *King (King,?)*, the model is capable of correctly answer *(King,Queen)*.



Try	-0.001388	0.031293	-0.524790	0.017237
to	0.319240	0.063160	-0.278580	0.261200
scroll	0.228140	-0.350200	-0.005245	0.137630
around	0.127130	0.087776	-0.174660	-0.006293
horizontally	0.545290	-0.446980	-0.167700	0.220940

each	-0.17246	0.054016	0.034461	0.19666
theme	0.46639	0.207400	0.185730	-0.40191
installation	0.49132	-0.048056	0.200700	-0.21108

is	-0.084961	0.50200	0.002382	-0.167550
true	0.096561	0.50832	-0.280250	-0.228730
in	0.089187	0.25792	0.262820	-0.029365

Firefox	0.245720	-0.300720	0.40298	-0.124060
launches	-0.147420	0.095634	0.15905	0.039806
without	0.009888	0.027157	-0.32870	-0.031992

window	0.687580	-0.345310	-0.18579	0.23870
sizes	0.182300	0.890110	-0.20703	-0.40545
Unspecified	0.091072	-0.006971	-0.19308	0.48667

tc_name				
TC_13_TRG	-0.022611	0.099271	-0.052506	-0.049523
TC_37_TRG	0.017220	0.163365	-0.108600	-0.033845
TC_60_TRG	0.114570	0.078011	-0.107812	-0.047740

br_name				
BR_1267501_SRC	0.066742	0.120455	-0.156679	0.029707

br_name	BR_1267501_SRC
tc_name	
TC_13_TRG	0.886369
TC_37_TRG	0.940672
TC_60_TRG	0.925198

Figure 2.10: Word Vector Example

For traceability recovery, word vectors can be used to measure the similarity between single words as exemplified above, and also between documents and queries the same way is made with IR techniques. The technique produces the document and query vectors, and the cosine similarity measure can be applied similarly to LSI and LDA.

A concrete example is used for illustrating the technique application, as we see in Figure 2.10. The same bug report and test cases from the previously explained techniques were reused. Our example is divided into five steps. In the first step (1), a word embedding with 300 dimensions and more than 1 million unique words was trained based on the Common Crawl Dataset – created based on texts from blogs, comments, and news in the web. In the next step (2), the tokens of each document (test case and bug report) were extracted – note we did not preprocess them –, and in the third step (3) the tokens of each document were grouped in a matrix of word vectors representing each document, the referred vectors in the matrix are a subset of the ones presented in the word embedding – for example, the test case 60 (*TC_60_TRG*) has the words *each*, *theme*, and *installation* and each one of them is represented by its 300-dimension vector. Next, in the fourth step (4) the average of the grouped vectors was calculated for each document (test case or bug report) and smaller matrices were created, each line of the matrix represents a document. In the last step (5), the cosine similarity was calculated for each pair of a test case and bug report.

Observe that the Word Vector technique correctly ranked the test cases in our example, identifying the test case 37 as the most relevant (see darker green cell) for our query (bug report), although the difference of the attribute similarity scores between each pair of documents is not so precise, considering that our scale (cosine similarity) is between -1 and 1. This may difficult the identification of correct and incorrect traces by the technique.

Chapter 3

Approach

The proposed approach uses IR and DL techniques as an *external/pluggable component*, in order to recover traceability links between bug reports, which are used as source artifacts, and test cases, used as target artifacts. During the analysis of an open-source project – the Mozilla Firefox –, we have observed the opportunity of using system features as intermediate artifacts, since they make the communication between the test and development teams easier, enforcing a common vocabulary.

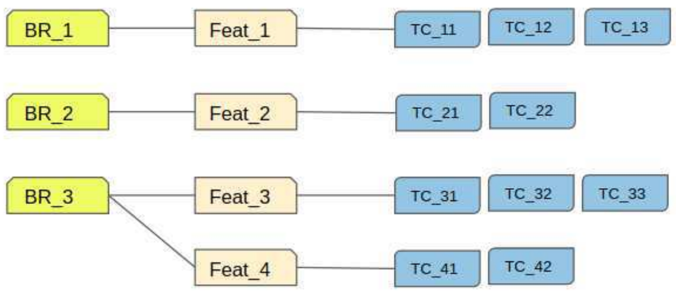


Figure 3.1: Bug Reports, System Features and Test Cases Relationships

Also, we have identified that most traceability links between bug reports and test cases cannot be recovered using only the information provided directly by the testers, once they do not create the links between the test cases and the bug reports as required during the testing period by the Mozilla's leading teams.

Analyzing the software artifacts organization into the project, we noticed if a bug report could be related to one specific feature, then it would be linked to the test cases of this feature. Figure 3.1 shows how these artifacts are related to each other, *BR_X* are bug reports, *Feat_Y* are system features, and *TC_YW* are test cases.

In our approach, as we see in Figure 3.2, the module *BR-TC Traces Builder* is responsible for recovering the trace links between bug reports and test cases using the selected techniques. The module receives a set of bug reports and maps them to a subset of the provided test cases by applying each IR and DL technique. As a result, we have a *Recovered BR-TC Trace Links Matrix* for each executed technique. The output of the module *BR-TC Traces Builder* is a binary matrix called *Recovered BR-TC Trace Links Matrix*, where each cell holds a value 1 – indicating the presence of a link between the test case (line) and bug report (column) – or a 0 – indicating the abscence of it. The *BR-TC Traces Builder* module is further detailed in the sequel.

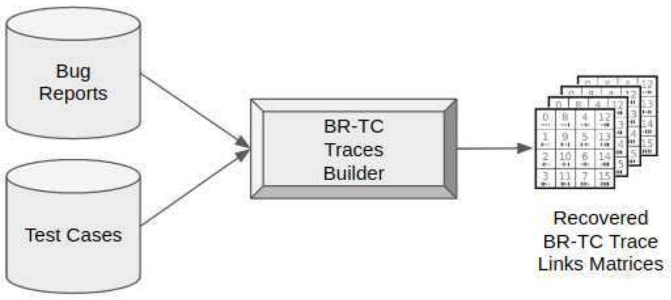


Figure 3.2: *BR-TC Traces Builder* Module

Figure 3.3 schematizes the *Traces Builder* in detail. As can be seen, it is composed of other two modules named *Traceability Engine* and *Trace Links Generator*. The *Traceability Engine* creates a similarity matrix from the input, where each column corresponds to a source artifact (bug report) and each line to a target artifact (test case). As explained in Chapter 2, the documents (test cases and bug reports) are vectorized and a representational matrix is built with the created vectors of each target document (test case) and then the source document (bug report) vector is compared with each test case vector through the cosine similarity score. At the end, we are able to create a similarity matrix for each pair of documents (bug report, test case) for each applied technique and rank the test cases based on a input bug report document.

In the created matrix, each cell holds a similarity score, calculated according to the applied technique (LSI, LDA, BM25 or Word Vector (WV)). The LSI similarity score $sim(d_j, q)$, for example, can be calculated with a document vector $d_j = (w_1, w_2, ..., w_N)$

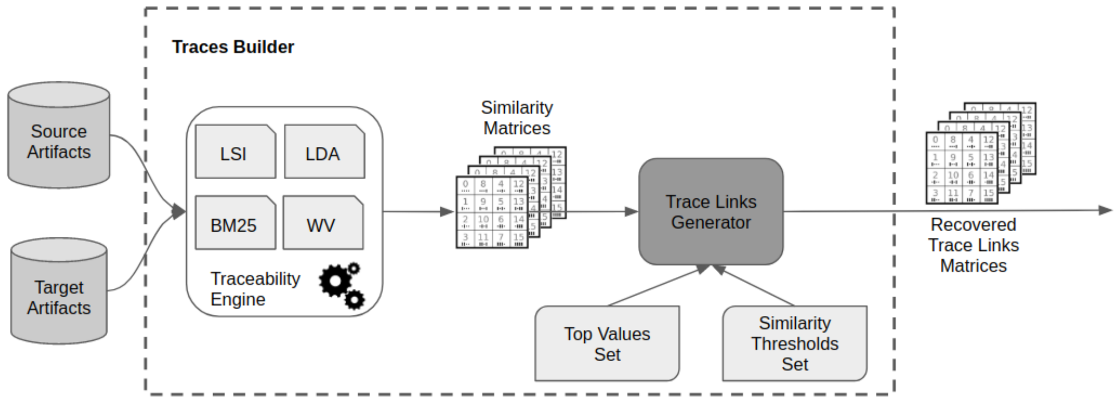


Figure 3.3: Traces Builder Submodules

and a query vector $q = (q_1, q_2, ..., q_N)$ as presented by Equation 3.1 [42; 6].

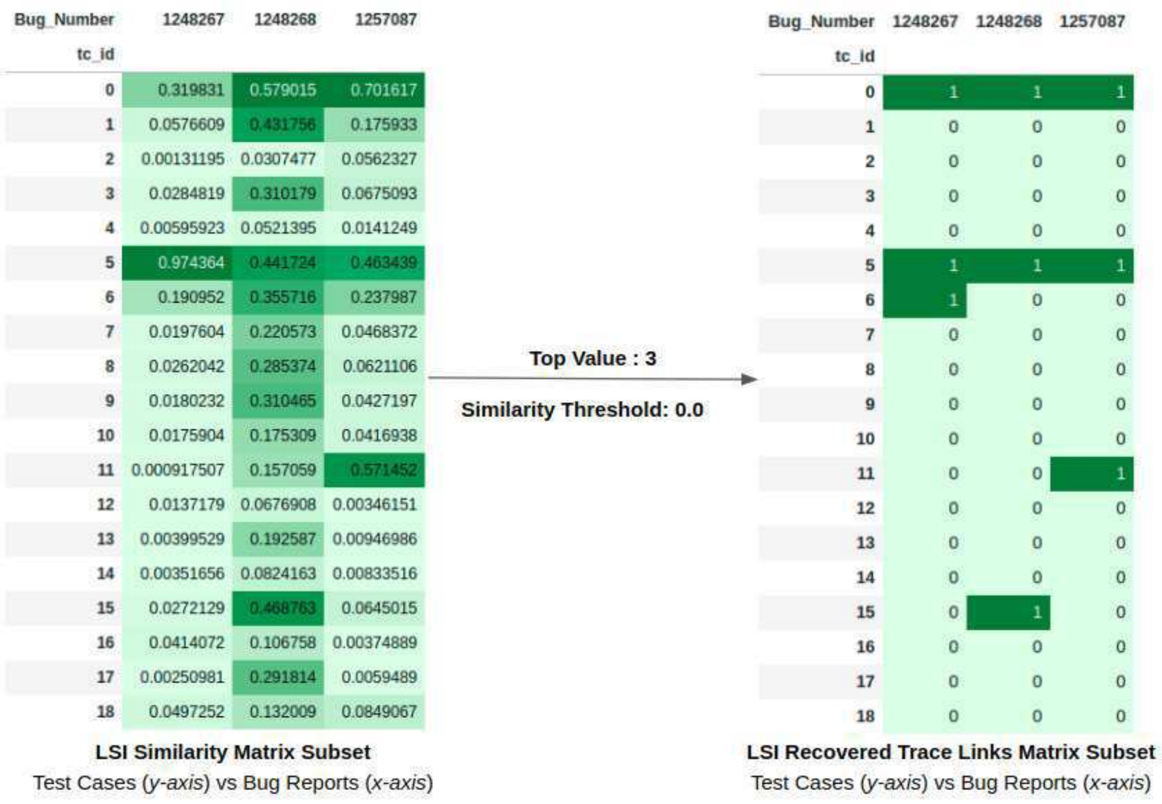
$$sim(d_j, q) = cos(d_j, q) = \frac{\sum_{i=1}^{N} w_i \cdot q_i}{\sqrt{\sum_{i=1}^{N} w_i^2 \cdot \sum_{i=1}^{N} q_i^2}} \tag{3.1}$$

Where $w_i(q_i) = f_i * idf_i$, f_i is the frequency of a term k_i in a document or query and idf_i is the inverse document frequency of k_i.

To the submodule *Trace Links Generator* is given three inputs: the set of similarity matrices generated by the *Traceability Engine*, a set of *Top Values*, and a set of *Similarity Thresholds*. The function combining the values of these two sets limits the number of documents returned to a query, in order to control the behavior of each technique when multiple sets of documents are recovered for each query, so the ranking capabilities of each technique can be evaluated [2; 9; 13; 19]. Top Values define absolute values of documents to be recovered; its possible values like TOP-1 – only the highest similarity score – and TOP-3 which returns the first three documents with the highest similarity scores.

Similarity thresholds designate a minimum similarity score between a document and a query that must be reached by a given technique. For example, (TOP-3, 0.85) states that only three documents (the three documents with the highest similarity values) must be recovered, with a similarity value higher than or equals to 0.85. This value defines a minimum similarity score, so each similarity matrix cell will be set as a positive trace link (1) or not (0). Therefore, as we see in Figure 3.3, for each combination of Top Value and Similarity Threshold an output matrix called *Recovered Trace Links Matrix* is created.

Figure 3.4 shows an example of the recovering of trace links from the LSI's similarity matrix for Top-3 and Similarity Threshold 0.0. Observe for each bug report (column) is returned a set of three test cases (line), corresponding to the highest similarity scores. In the matrix in the right side of the figure, the positive (returned) traces are depicted as a one (1) and highlighted in dark green, while the negative (0) traces are colored in light-green.



Bug_Number	1248267	1248268	1257087
tc_id			
0	0.319831	0.579015	0.701617
1	0.0576609	0.431756	0.175933
2	0.00131195	0.0307477	0.0562327
3	0.0284819	0.310179	0.0675093
4	0.00595923	0.0521395	0.0141249
5	0.974364	0.441724	0.463439
6	0.190952	0.355716	0.237987
7	0.0197604	0.220573	0.0468372
8	0.0262042	0.285374	0.0621106
9	0.0180232	0.310465	0.0427197
10	0.0175904	0.175309	0.0416938
11	0.000917507	0.157059	0.571452
12	0.0137179	0.0676908	0.00346151
13	0.00399529	0.192587	0.00946986
14	0.00351656	0.0824163	0.00833516
15	0.0272129	0.468763	0.0645015
16	0.0414072	0.106758	0.00374889
17	0.00250981	0.291814	0.0059489
18	0.0497252	0.132009	0.0849067

LSI Similarity Matrix Subset
Test Cases (*y-axis*) vs Bug Reports (*x-axis*)

Bug_Number	1248267	1248268	1257087
tc_id			
0	1	1	1
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	1	1	1
6	1	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	1
12	0	0	0
13	0	0	0
14	0	0	0
15	0	1	0
16	0	0	0
17	0	0	0
18	0	0	0

LSI Recovered Trace Links Matrix Subset
Test Cases (*y-axis*) vs Bug Reports (*x-axis*)

Figure 3.4: Traces Recovering Process Example

Chapter 4

Building an Oracle Matrix

In this chapter, we explain the process of manually creating the oracle matrix. This matrix maps bug reports to system features of the Mozilla Firefox, such as was explained in the previous Chapter 3 with the help of a crowdsourcing application to gather the answers from volunteers and a researcher through a survey. The built oracle is need for the execution of the empirical study (case study) described at Chapter 5.

4.1 Context

The Mozilla Firefox internet browser [1] is a real, extensive and active open-source project developed by the Mozilla Corporation. The Mozilla's development team uses the Rapid Release (RR) development model [29], in which they select a set of features for testing during a *TestDay* at the end of each sprint (Each Firefox release has two or three testdays). After all test cases for the features under test are executed, the set of bug reports is recorded. Figure 4.1 details the RR development model with three released versions of Mozilla Firefox, highlighting the TestDays (*TDX*), the features tested in each TestDay (*Feat_Y*) and the test cases of each feature (*TC_W*).

Core members of the Mozilla's QA team organize TestDay data into an open-access Etherpad[2] online document, containing the specification of features to test, test cases associated with each feature, and the set of bug reports fixed by developers during the sprint and

[1]https://www.mozilla.org

[2]https://public.etherpad-mozilla.org/

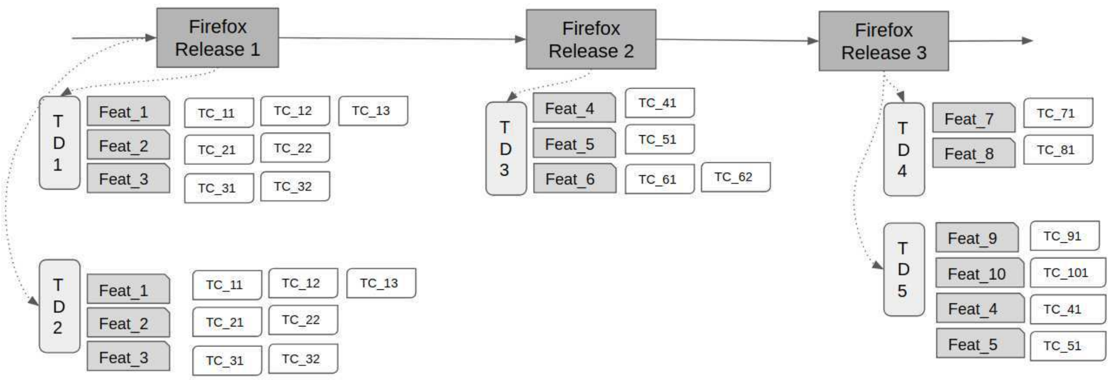


Figure 4.1: Mozilla Firefox Development Model: Rapid Release (RR)

need to be checked in that TestDay. By the end of a TestDay, each test case in the document is specified with keywords PASS or FAIL. When a test case fails, the tester is advised to create a bug report in Bugzilla[3] and create a link in the etherpad document as the result of the failed test case for later traceability.

However, testers often either do not create the links as required or do not create the bug report at all, then several test cases marked as failed have no associated bug reports. Most traceability links between bug reports and test cases cannot be recovered using the information provided directly by the testers. Seeking to solve this problem, we saw the possibility of using system features as an intermediate artifact to link bug reports and test cases. If a bug report is related to one specific feature, then it links to the test case of this feature.

4.2 Participants

We recruited volunteers to, based on the reading of the Mozilla's documentation, point out which Firefox features they think a given bug report is related. As a result, they produced a *matrix of traceability links* (oracle) between features and bug reports, as a first step to relate bug reports to test cases. This step was needed for scalability since there are much more test cases (195) than features (19) and relate bug reports directly with the test cases would require a large amount of manual work, time and resources which were not available.

A total of nine volunteers participated in the study, who were recruited by e-mail invita-

[3]http://bugzilla.mozilla.org

tion. All volunteers have a Bachelor degree in Computer Science; while one holds a Ph.D., another one is a full-time software developer, and seven are master students. They all have professional experience in software development, including knowledge about key concepts involved in software development and testing (such as system features, test cases, and bug reports). Previously to the volunteers‘ participation, the researcher (also named expert) – who had previous knowledge of the Firefox features, test cases, bug reports, and the traceability process – also responded to the same tasks of the volunteers and another *matrix of traceability links* (oracle) was generated from his answers.

4.3 Used Data

The used data set of test cases and system features was extracted from Firefox TestDays from 2016/06/03 to 2017/01/06. Test cases were frozen in this period, which is appropriate for our analysis, once the test cases do not evolve in this time interval. A total of 195 test cases were manually collected from this period – 12 TestDays. We identified a set of 19 different Firefox features tested during this period. Each test case is associated with one specific Firefox feature and is explicitly indicated by the Mozilla's QA Team in the TestDay available documents. Table 4.1 shows the Firefox Features used, the particular Firefox versions as well as the number of test cases associated with each feature. The features are used to aggregate the test cases of a TestDay, so each test case executed in a TestDay is related to one of the tested features.

Furthermore, we employed the following criteria to select a total of 93 bug reports from a set of +35000 bugs collected from Bugzilla updated between 2016/06/01 and 2018/12/31:

- Firefox version must be between 48 to 51;
- Status must be RESOLVED or VERIFIED;
- Priority must be P1, P2 or P3, the highest priority levels;
- Resolution field must be FIXED, which means the bug was already fixed when collected for our study;

Table 4.1: Firefox Features

Feature Name	**Firefox Version**	**TCs Amount**
New Awesome Bar	48 and 50	13
Windows Child Mode	48	11
APZ - Async Scrolling	48	22
Browser Customization	49	6
PDF Viewer	49	8
Context Menu	49	31
Windows 10 Compatibility	49	6
Text to Speech on Desktop	49	2
Text to Speech in Reader Mode	49	8
WebGL Compatibility	49	3
Video and Canvas Renderization	49	2
Pointer Lock API	50	11
WebM EME support for Widevine	50	6
Zoom Indicator	51	21
Downloads Dropmaker	51	18
WebGL2	51	3
FLAC support	51	6
Indicator for device permissions	51	16
Flash support	51	2

- Severity must be "major," "normal," "blocker," or "critical," ruling out "enhancements";

The Status[4] field indicates the current state of a bug. The states can change accordingly to a predefined state machine. Only specific status transitions are allowed. The Resolution field indicates what happened to this bug if it was fixed or not. These filters reduced the number of bugs to be analyzed by the volunteers in the study. Also, these filters allow selecting a subset of bug reports that are the most relevant in the entire dataset, following these criteria

[4]Bug Fields: https://bugs.documentfoundation.org/page.cgi?id=fields.html

stated by the community, and that is related to the test cases we examined.

4.4 Procedure

The study was executed following the process depicted in Figure 4.2.

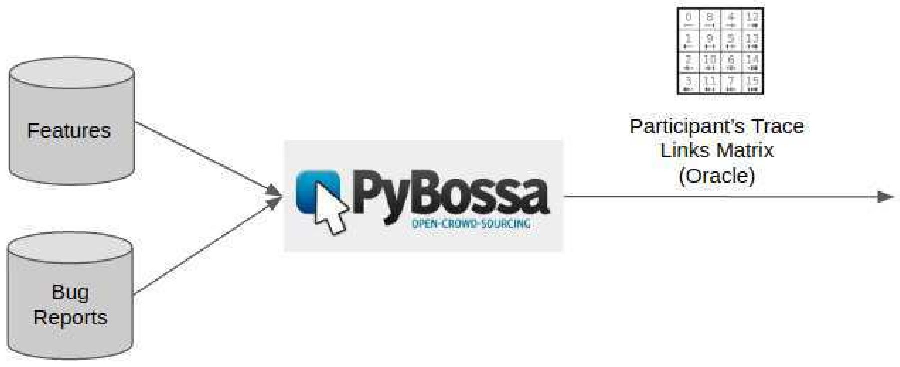


Figure 4.2: Scheme of First Empirical Study – Oracle Creation

As the input of the scheme, two datasets of System Features and Bug Reports feed the *PyBossa Platform*[5], which hosts the web applications to support the participation of volunteers and the expert. The hosted applications collect the answers to a set of tasks proposed to the participants. At the end of the process, after processing these answers, we have a manually generated oracle (*Participant's Trace Links Matrix*). Further details on this process are provided in the rest of this section.

As already explained, we used the PyBossa crowdsourcing platform to coordinate the participation of each volunteer and aggregate his/her contributions; in this environment, is defined an application or project which hosts a set of tasks. Each one of these tasks is very specific, for example, define if a reported issue in a Bug Tracking System is a bug report or a change request, tagging it as belonging to one of two classes: BUG_REPORT or CHANGE_REQUEST. This kind of task is relatively simple for a human analyst to accomplish, but it is very hard for a machine. Once finished the resolution of many tasks by humans, the data set of tagged issues can be used for the training of machine learning models in order to recognize issues as bug reports or change requests automatically, for example.

[5]PyBossa Platform: https://pybossa.com/

In our study, we created a set of 93 tasks, one for each one of the 93 bug reports and two identical versions of these tasks were deployed to the volunteer's and the expert's applications, so the answers could be collected. The workspace included the complete bug report information, including the first comment made by the bug reporter, generally detailing the steps for reproduction, along with a checklist with the 19 features targeted. We decided to consider only the first comment, once the presence of noisy text – from the discussions between the many involved people in the Bugzilla – can difficult the technique's effectiveness into doing the traceability later.

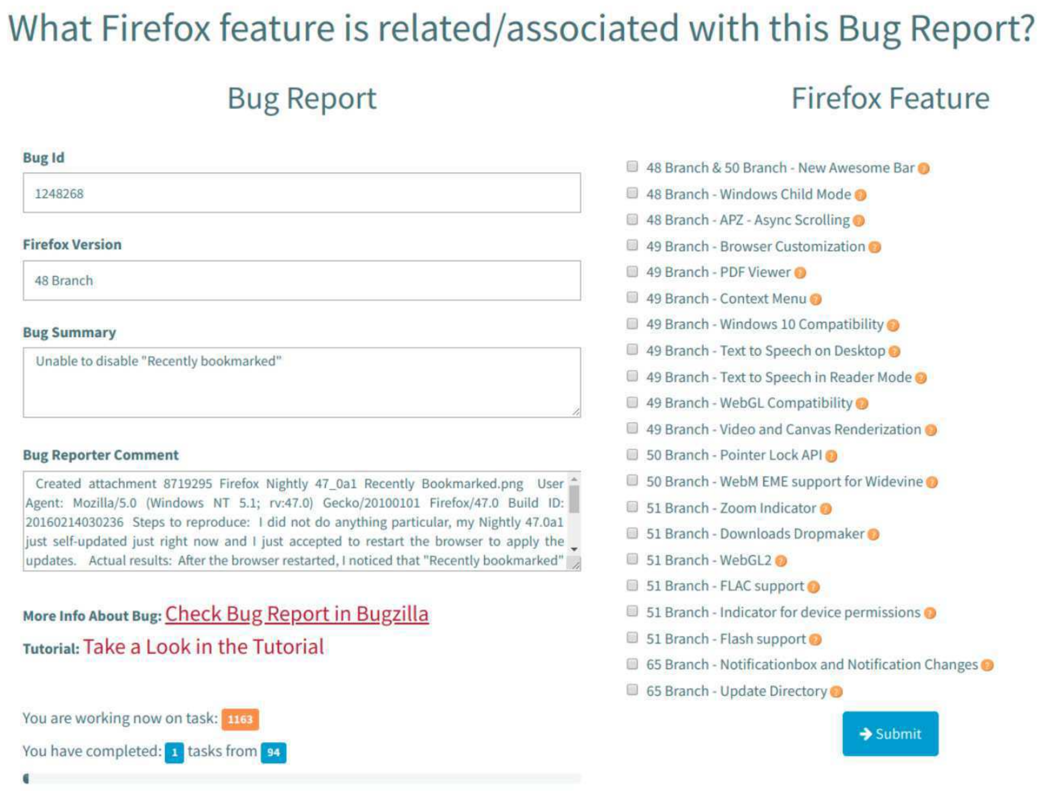
The task of the participants consisted in reading the bug report and the features descriptions, and thus decide which ones, if any, were related to that specific bug report. Additionally, we provided a tutorial made for the application as well as links to the original description of the bug report in the Bugzilla and additional information about the features[6], in the case the participants had doubts about the system features. Figure 4.3 shows a screenshot of the volunteers' application in the PyBossa platform.

All volunteers watched a 10 minutes presentation about the targeted Firefox features and the PyBossa workspace. They had access to the training material during the execution of the tasks. The study was carried out with each volunteer individually, during a scheduled session of 20 minutes, when each volunteer contributed with around ten tasks. We considered a feature to be related or associated with a given bug report if *at least one* of the following conditions is satisfied:

- the bug report summary (title) or the bug report first comment (steps to reproduce) cites the feature(s) directly;
- the bug report directly impacts any of the listed features.

If a participant detected any of the two cases above, he/she should indicate the existing (positive) relationship in the application's task submission. The positive relationship indi-

[6]https://support.mozilla.org
https://wiki.mozilla.org/QA/
https://www.paessler.com/manuals
https://addons.mozilla.org
https://developer.mozilla.org


What Firefox feature is related/associated with this Bug Report?

Bug Report

Bug Id

1248268

Firefox Version

48 Branch

Bug Summary

Unable to disable "Recently bookmarked"

Bug Reporter Comment

Created attachment 8719295 Firefox Nightly 47_0a1 Recently Bookmarked.png User Agent: Mozilla/5.0 (Windows NT 5.1; rv:47.0) Gecko/20100101 Firefox/47.0 Build ID: 20160214030236 Steps to reproduce: I did not do anything particular, my Nightly 47.0a1 just self-updated just right now and I just accepted to restart the browser to apply the updates. Actual results: After the browser restarted, I noticed that "Recently bookmarked"

More Info About Bug: Check Bug Report in Bugzilla

Tutorial: Take a Look in the Tutorial

You are working now on task: 1163

You have completed: 1 tasks from 94

Firefox Feature

- ☐ 48 Branch & 50 Branch - New Awesome Bar
- ☐ 48 Branch - Windows Child Mode
- ☐ 48 Branch - APZ - Async Scrolling
- ☐ 49 Branch - Browser Customization
- ☐ 49 Branch - PDF Viewer
- ☐ 49 Branch - Context Menu
- ☐ 49 Branch - Windows 10 Compatibility
- ☐ 49 Branch - Text to Speech on Desktop
- ☐ 49 Branch - Text to Speech in Reader Mode
- ☐ 49 Branch - WebGL Compatibility
- ☐ 49 Branch - Video and Canvas Renderization
- ☐ 50 Branch - Pointer Lock API
- ☐ 50 Branch - WebM EME support for Widevine
- ☐ 51 Branch - Zoom Indicator
- ☐ 51 Branch - Downloads Dropmaker
- ☐ 51 Branch - WebGL2
- ☐ 51 Branch - FLAC support
- ☐ 51 Branch - Indicator for device permissions
- ☐ 51 Branch - Flash support
- ☐ 65 Branch - Notificationbox and Notification Changes
- ☐ 65 Branch - Update Directory

Figure 4.3: Volunteers's application in PyBossa platform

cates an existing trace link between two artifacts, in this case between a bug report and a system feature.

4.5 Results

The oracle generated based on the volunteers' answers and the one generated based on the expert's answers were surprisingly different. The volunteer's oracle indicated the existence of 93 positives links between bug reports and system features, while the expert's oracle indicated only 58. This considerable difference leads us to investigate different options of oracle we could use and which one would be more reliable:

1. the oracle generated by the expert (*Expert-Only*);
2. the oracle generated by the volunteers (*Volunteers-Only*);
3. the intersection of expert's and volunteers' oracles (*Exp-Vol-Intersection*);

4. the union of expert's and volunteers' oracles (*Exp-Vol-Union*).

Figure 4.4 shows the distribution of the number of system features (y-axis) by bug reports (x-axis) for different strategies.

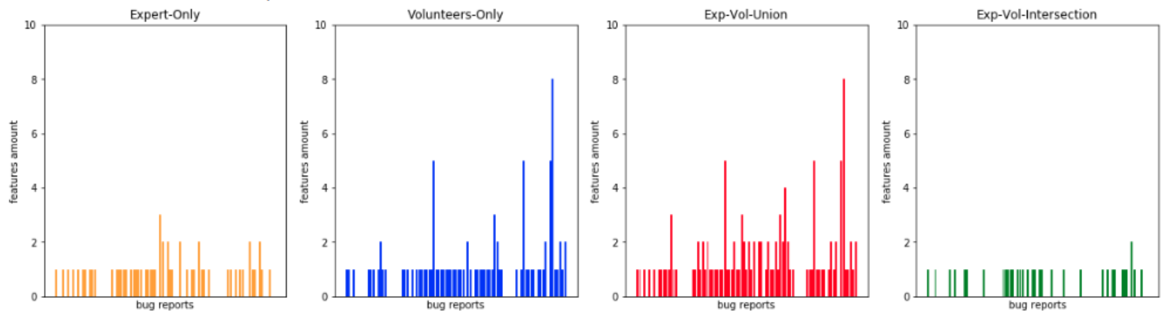


Figure 4.4: Amount of features by bug report

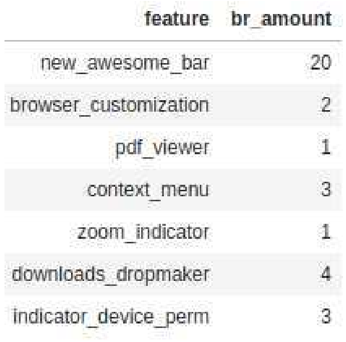
We can see the mean amount of features (μ) by bug report in the Expert-Only strategy ($\mu_{Exp} = 0.64$) is smaller than in the Volunteers-Only ($\mu_{Vol} = 1.0$) strategy. For the Exp-Vol-Union strategy, 1.3 features are attributed in mean for each bug report ($\mu_{Union} = 1.3$), while in the Exp-Vol-Intersection strategy this amount is 0.37 ($\mu_{Inter} = 0.37$).

We see the volunteers tended to point out traceability links when they had doubts about its existence. An example of this, is the bug report *1432915*, whose title is *Do not write the kMDItemWhereFroms xattr metadata for files downloaded in Private Browsing mode*. The expert pointed out the bug report as related only with the *Downloads Dropmaker* feature, while the volunteers pointed out as related with the *Download Dropmaker* and *New Awesome Bar* features.

The value of Cohen's Kappa coefficient (κ) for inter-rater agreement between the volunteers and the expert is $\kappa = 0.426$ with is considered a weak inter-rater agreement level ($0.40 \leq \kappa \leq 0.59$) [30]. This result indicates that the expert and volunteers do not agree about the existence of many traces.

Continuing the analysis, we checked the intersection between volunteers' and the expert's oracles had only 34 traces. We focused in the intersection, once we have two disjoint sets of answers from the volunteers and the expert: the one they agree (34 traces) and another one they do not agree (59 traces). Investigating the traces they do not agree, we observed the existence of wrong traces indicated by the volunteers. One example of that is the bug report

1287687, that is relative to user's data synchronization with Firefox's cloud system, and the volunteers incorrectly indicated as linked with the system feature *APZ Async Scrolling*. We attribute this error to the lack of experience of the volunteers and consequential failure in identifying the type of synchronization that is made in the *APZ* system feature.



feature	br_amount
new_awesome_bar	20
browser_customization	2
pdf_viewer	1
context_menu	3
zoom_indicator	1
downloads_dropmaker	4
indicator_device_perm	3

Figure 4.5: Amount of bug reports per system feature

One second example of mistake is the bug report *1290424* which is relative to the *New Awesome Bar* but was indicated by the volunteers as relative to *Indicator for Device Permission*. Again, the lack of understanding about the features may be the cause of the mistakes. The fact the volunteers did not belong to Mozilla's development or testing teams may be the main cause of these mistakes. The amount of bug reports per system feature is detailed in Figure 4.5. Note that only seven features had bug reports linked to them, the remaining features had none.

Due to the presence of these errors made by the volunteers and how we can not adopt only the expert's answers (this would bias this work), we decided to use the **intersection** of expert's and volunteers' oracles as our reference oracle for judging the effectiveness of the techniques, so we have an agreement between the two raters (expert and volunteers) and a more reliable oracle for evaluating the techniques.

Chapter 5

Empirical Study - Bug Reports to Test Cases Traceability

This study focuses on the traceability between the bug reports and the test cases, that uses the oracle that was produced as explained in the previous chapter. This study was conducted as a Case Study based on the Mozilla Firefox available bug reports, test cases, and system features.

5.1 Study Definition and Context

We aim to evaluate the *BR-TC Traces Builder*'s trace links matrices generated in relation to an oracle (from bug reports to test cases) derived from the volunteers' and expert's participation in the first study creating the first oracle (from features to bug reports). This study aims at discussing answers to the following research question: *RQ1: Which technique presents the best effectiveness in the context of traceability between bug reports and test cases?* in terms of $Precision$, $Recall$, and F_2-$Score$.

5.2 Objects

We used the same bug reports and features from the first empirical study and the same test cases from where the features were extracted. We used LSI, LDA, and BM25 as IR techniques, and Word Vector as the DL technique. The values of *Similarity Thresholds* were the range [0.0, 0.1, ..., 0.9], so we could address the techniques effectiveness variation considering each cut value based on the achieved similarity between the bug report and test case. We have used the values 10, 20 and 40 as *Top Values*, once the average number of test cases linked with bug reports is not larger than 40, and we could also address the techniques effectiveness variation considering each of these fixed cuts. We expect some combinations of Top Values and Similarity Thresholds may have better effectiveness than others, boosting each technique performance in terms of the studied metrics.

5.3 Study Procedure

The Figure 5.1 shows a schematization of this second study in detail.

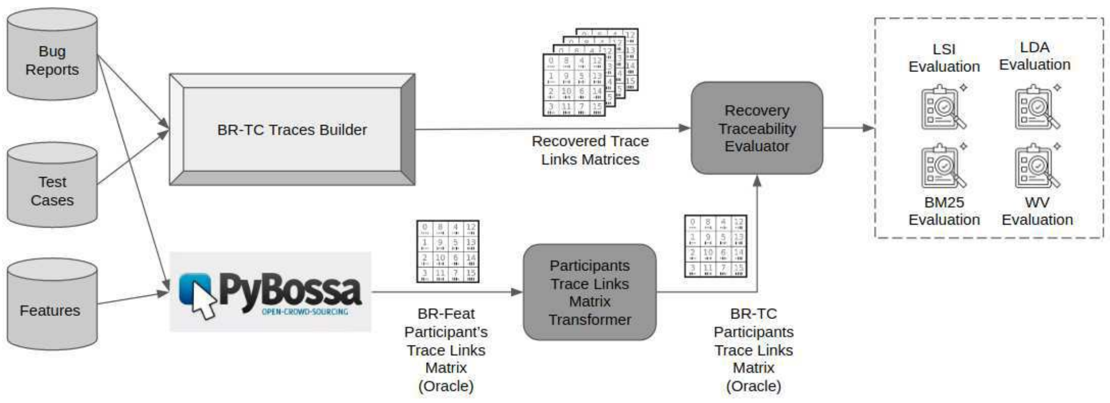

Bug Reports
Test Cases
Features
BR-TC Traces Builder
PyBossa
OPEN-CROWD-SOURCING
Recovered Trace Links Matrices
Recovery Traceability Evaluator
LSI Evaluation
LDA Evaluation
BM25 Evaluation
WV Evaluation
BR-Feat Participant's Trace Links Matrix (Oracle)
Participants Trace Links Matrix Transformer
BR-TC Participants Trace Links Matrix (Oracle)

Figure 5.1: Scheme of Second Empirical Study

As the input of the scheme, we see three data sets of System Features, Test Cases, and Bug Reports entering the *BR-TC Traces Builder* and the *PyBossa Platform*. The first, as explained in Chapter 3, produces binary matrices for each combination of Top Value and Similarity Threshold for each technique. On the other hand, the second provides the oracle generated through the volunteers' and expert's participation (see Chapter 4). Then, the module *Recovery Traceability Evaluation* receives both outputs and evaluate the effectiveness of the techniques.

Observe the presence of the component *Participants Trace Links Matrix Transformer*, it is responsible for transforming the oracle, which maps bug reports to features – obtained in the first study –, into an oracle that maps bug reports to test cases.

5.3.1 BR-TC Traces Builder Evaluation

The oracle between bug reports and test cases is used as input to the next phase, the evaluation of the techniques. The *BR-TC Traces Builder* received as input bug reports and test cases from the Mozilla's Firefox original documentation. Since the *BR-TC Traces Builder* is responsible for producing *recovered trace links matrices* between the bug reports and test cases used, each line of the produced matrix represents a test case, whereas each column represents a bug report.

Next, the set of trace links matrices recovered by the *BR-TC Traces Builder* is passed to a module (*Recovery Traceability Evaluator*) that evaluates the effectiveness of each technique applied in comparison to the *Trace Links Matrix (Oracle)* built by the participants. For each used IR and DL technique, the values of each technique's parameter are defined according to the literature's recommendations and to the parameter's combinations that best perform for the data set of bug reports and test cases.

The *BR-TC Traces Builder*'s *Traceability Engine* processes the input artifacts in two phases: preprocessing and execution. For Preprocessing, we used Python's NLTK[1] (Natural Language Toolkit), a well-established platform for natural language processing (NLP) applications. It was applied for *tokenization*, stop-word removal and stemming/lemmatization, required by the IR techniques.

Regarding execution of the IR and DL techniques, we applied open-source implementations of the chosen IR and DL techniques – *Scikit-Learn Data Analysis Toolkit*[2] (LSI, LDA), the *Gensim Library*[3] (BM25), and the *SpaCy Library*[4] (Word Vector). The scripts are available online[5].

SciKit Learn's LSI and LDA techniques require a vectorizer which is responsible among other things for tokenizing the documents. We have used the `TfidfVectorizer` implementation provided by the own framework and the NLTK's English stopwords. Besides that, the LSI requires the number of components for making the dimensionality reduction and the LDA requires the number of topics. A parameter search performed showed which parame-

[1]https://www.nltk.org

[2]SciKit: https://scikit-learn.org/stable/

[3]Gensim: https://radimrehurek.com/gensim/

[4]SpaCy: https://spacy.io/

[5]https://doi.org/10.5281/zenodo.2643447

ters best fit the data set: we tested the techniques with smaller and greater parameter values (5,10,20,40,100), but coincidentally 20 was the best value for both LSI and LDA techniques.

BM25's implementation was executed with the recommended values for English texts [38; 7]; k_1 (the effect of term frequency over) is 1.2, whereas b (the effect of the document length) is 0.75. The BM25 scoring function's output values are outside the scale [0,1] and need to be normalized, so we used the SciKit Learn's `MinMaxScaler` to fit the similarity values in the scale [0,1] and then we could evaluate the techniques using the variable similarity threshold in the scale [0,1]. Finally, for the Word Vector implementation, we used a pre-trained neural network (word embedding) called GloVe (Global Vectors for Word Representation)[6] [36], resulting in a model of 631 MB, based on a vector space representation of +1 million tokens with 300 dimensions[7] extracted from blogs, news, and comments in the web in general.

Following the preprocessing phase the techniques are executed with the tokenized bug reports and test cases, similarity matrices (see Chapter 2) were then generated, and different *BR-TC Recovered Trace Links Matrices* were created according to multiple combinations of *Top Values* and *Similarity Thresholds*. For Top Values, we used 10, 20, and 40 – so a technique could recover all test cases linked with a bug report –, and a range of similarity threshold values between 0.0 and 0.9 (included) with a step size of 0.1 (0.0, 0.1, ..., 0.9), which is compatible with the range of the values into the similarity matrices that interest us (the ones with similarity greater than zero, meaning closest similarity between the documents). Finally, the *Recovered Traceability Evaluator* assessed each technique using selected metrics and the participant's trace links matrices (oracles).

5.4 Research Method

5.4.1 Metrics

$Precision$, $Recall$ and F_2-$Score$ are very common metrics used in the field of traceability recovery [22] and are defined as follows:

[6]https://nlp.stanford.edu/projects/glove/

[7]https://spacy.io/models/en

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

$$F_\beta\text{-}Score = (1 + \beta^2).\frac{Precision * Recall}{(\beta^2 * Precision) + Recall} \tag{5.3}$$

Where TP is the number of True Positives, FP is the number of False Positives and FN is the number of False Negatives. The F_β-$Score$ is a general version of the F-$Score$, and the F_2-$Score$ ($\beta = 2$) is an unbalanced version of F_1-$Score$. The latter (F_1-$Score$) attributes equal importance to $Precision$ and $Recall$ scores, while the first (F_2-$Score$) attributes the double of the importance to the $Recall$ score over the $Precision$ score [3].

Remembering Figure 4.2, the ***Recovery Traceability Evaluator*** takes each ***BR-TC Recovered Trace Links Matrix*** (RTM_i) from the set of recovered matrices (RTM) and compares with the ***BR-TC Volunteers Trace Links Matrix (Oracle)*** – derived from the volunteers' answers – producing a triple with the $Precision$, $Recall$, and F_2-$Score$ ($P_{RTM_i}, R_{RTM_i}, F_{RTM_i}$) measures for each one of them. For each different technique, it is calculated the mean value of each metric.

5.4.2 Recovery Effort Index – *REI*

The metric Recovery Effort Index was proposed by Antoniol *et al.*[2] in order to estimate the amount of effort required to manually analyze the results of a traceability recovery technique, discarding the false positives, when comparing to completely manual analysis. Inspired by their work, we used a free adaptation of their metric focusing on the multiple combinations of Top Values and Similarity Threshold presented in our study; in our version, we calculated REI for each combination of Top Value and Similarity Threshold for each technique and compared the obtained $Precision$ for that case with the $Precision$ obtained by the oracle created only by the volunteers in relation to an oracle created by an Expert (see Chapter 3). The REI value associated with a technique is the mean of all calculated REI's. The Equation 5.4 shows the REI formula.

$$REI_T = \frac{\sum_{i,j} \frac{OrcVolPrec}{T_{i,j}Prec}}{|S_{i,j}|} \tag{5.4}$$

Where REI_T is the REI of a technique T, $OrcVolPrec$ is volunteer's oracle $Precision$, $T_{i,j}Prec$ is the $Precision$ of a technique with Top Value i and Similarity Threshold j, and $|S_{i,j}|$ is the cardinality of the set of combinations of Top Values and Similarity Thresholds.

5.4.3 Goodness

Additionally, we discuss the obtained results of $Precision$ and $Recall$ based on a scale of ***Goodness*** of traceability recovery defined by Hayes *et al.* [21], which establish some boundaries for these metrics to classify the level of traceability recovery as Acceptable, Good or Excellent. Table 5.1 details these boundaries. Additionally, we used the reference values in the scale to estimate the level of *Goodness* in relation to the F_2-$Score$ metric, identically to what was made by Merten *et al.* [31].

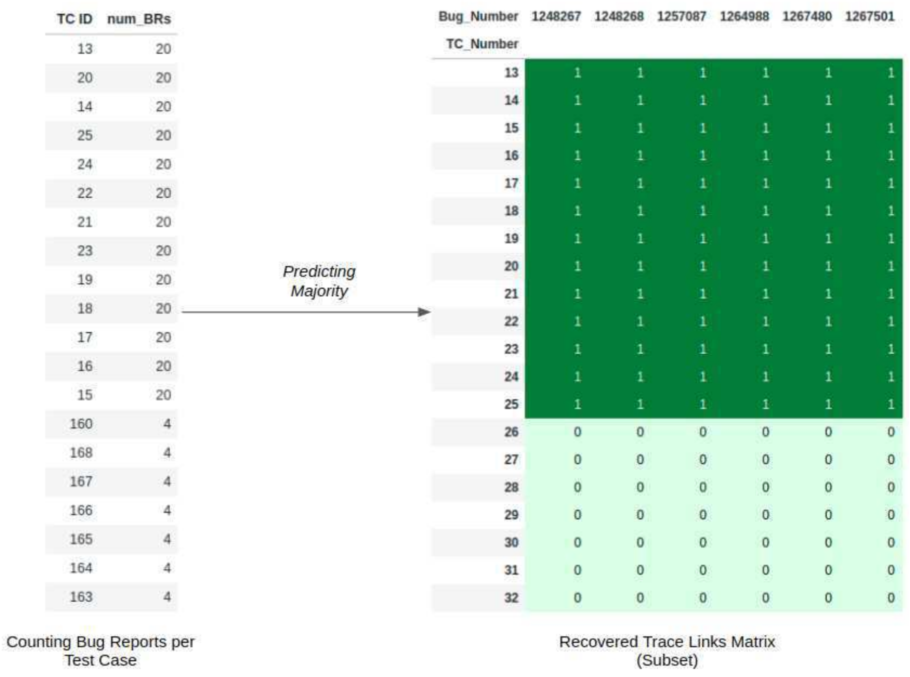
Table 5.1: Goodness Level

Measure	**Acceptable**	**Good**	**Excellent**
$Recall$	>60%	>70%	>80%
$Precision$	>20%	>30%	>50%
F_2-$Score$	>42.85%	>55.26%	>66.66%

5.4.4 ZeroR Predictor

In order to have a baseline of comparison, we implemented a *ZeroR Predictor/Classifier* [40] to classify a candidate trace between a bug report and test case as existent (1) or not existent (0). A *ZeroR Classifier* simply predicts the majority class, an example of the generated predictions are depicted in Figure 5.2.

On the left side of the figure, we see the counting of each test case (*TC ID*), which are our target artifacts, and that the test cases with the greatest number of bug reports (*num_BRs*) related to it (from the oracle) have 20 bug reports each. In this case, we have more than one *majority class* and the classifier predicts all of them as one major class. We can observe the


Predicting Majority
Counting Bug Reports per Test Case
Recovered Trace Links Matrix (Subset)

TC ID	num_BRs
13	20
20	20
14	20
25	20
24	20
22	20
21	20
23	20
19	20
18	20
17	20
16	20
15	20
160	4
168	4
167	4
166	4
165	4
164	4
163	4

Bug_Number	1248267	1248268	1257087	1264988	1267480	1267501
TC_Number						
13	1	1	1	1	1	1
14	1	1	1	1	1	1
15	1	1	1	1	1	1
16	1	1	1	1	1	1
17	1	1	1	1	1	1
18	1	1	1	1	1	1
19	1	1	1	1	1	1
20	1	1	1	1	1	1
21	1	1	1	1	1	1
22	1	1	1	1	1	1
23	1	1	1	1	1	1
24	1	1	1	1	1	1
25	1	1	1	1	1	1
26	0	0	0	0	0	0
27	0	0	0	0	0	0
28	0	0	0	0	0	0
29	0	0	0	0	0	0
30	0	0	0	0	0	0
31	0	0	0	0	0	0
32	0	0	0	0	0	0

Figure 5.2: ZeroR Classifier Predictions

results of the predictions in the *Recovered Trace Links Matrix* on the right side of the figure. Note how the classifier attributed 1 to the test cases with 20 bug reports, and 0 to the other test cases.

5.5 Results and Discussion

5.5.1 Oracle Generation

Before starting the analysis of the effectiveness of the techniques, we evaluated the generated oracles based on different strategies, the same way we did previously (see Chapter 4). The referred strategies are:

- oracle generated only from the volunteers' oracle (bug reports to features) *(Vol-Only)*;
- oracle generated only from the expert's oracle (bug reports to features) *(Exp-Only)*;
- intersection between the *Vol-Only* oracle and *Exp-Only* oracle *(Exp-Vol-Intersection)*;
- union between the *Vol-Only* oracle and *Exp-Only* oracle (*Exp-Vol-Union*).

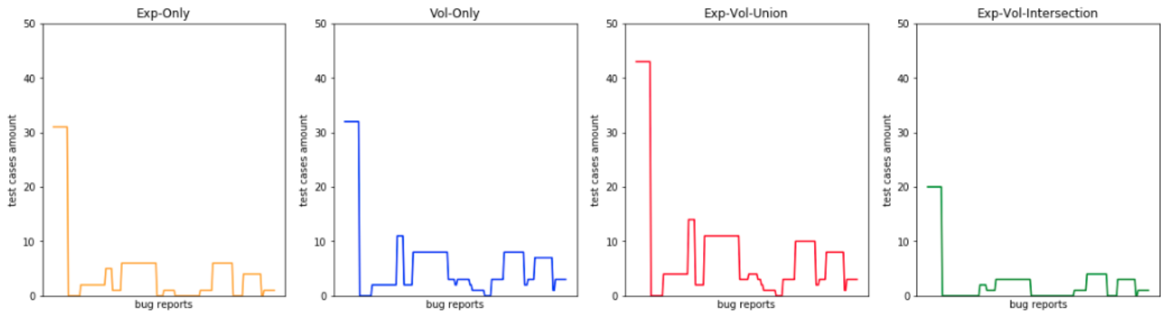


Figure 5.3: Number of test cases by bug report

Figure 5.3 shows the distribution of the number of test cases by bug report for each studied strategy. Analyzing the presented distributions, we see the *Exp-Only* strategy has a mean number of test cases per bug report (μ) smaller than the *Vol-Only* strategy ($\mu_{Exp} = 4.5$ and $\mu_{Vol} = 6.2$ respectively). As expected, the *Exp-Vol-Union* strategy has the highest mean ($\mu_{Union} = 8.0$) and the *Exp-Vol-Intersection* strategy the lowest ($\mu_{Inter} = 2.6$).

When we compare Cohen's kappa coefficients considering the *Exp-Only* and *Vol-Only* strategies, we observe that $\kappa = 0.4638$. This value indicates a weak inter-rater agreement level ($0.40 \leq \kappa \leq 0.59$) [30], and we can conclude the expert and volunteers do not agree about the existence of many traces.

In reason of the low level of agreement between the expert and the volunteers, we decided to analyze the effectiveness of the techniques using the ***intersection*** of the volunteers' and expert's oracles. As in the first study, we chose the intersection strategy, because it produces a more reliable oracle – built from the agreement between the answers of the volunteers and the expert.

The volunteers indicated the existence of 1205 positive links (traces) between bug reports and test cases, whereas the expert indicated 874. The intersection of the answers had 514 traces. Through the exploration of the remaining 691 traces recovered by the volunteers, we discovered similar mistakes as in the previous chapter. One example of that is the bug report *1306639* which is relative to the system feature *New Awesome Bar* and its telemetry recordings but was indicated by the volunteers as related to test cases from *Context Menu* system feature.

The intersection oracle traces are distributed as indicated in Table 5.2. Note that only seven features do appear in it, this is due to the first oracle which only had these seven fea-

Table 5.2: Number of traces in intersection oracle grouped by system feature

System Feature	num_BRs	num_TCs	num_Traces
New Awesome Bar	20	13	260
Browser Customization	2	6	12
PDF Viewer	1	8	8
Context Menu	3	31	93
Zoom Indicator	1	21	21
Downloads Dropmaker	4	18	72
Indicator for Device Permissions	3	16	48

tures presenting positive traces after the ***intersection*** operation between volunteers' answers and the expert's answers. The column **num_TCs** refers to the number of test cases that one system feature has in the Mozilla's documentation, the **num_BRs** refers to the number of bug reports related to that specific system feature, and the **num_Traces** to the number of traces (**num_BRs** * **num_TCs**).

5.5.2 General Evaluation

Figure 5.4 presents a bar chart with the obtained results for each applied IR or DL technique.

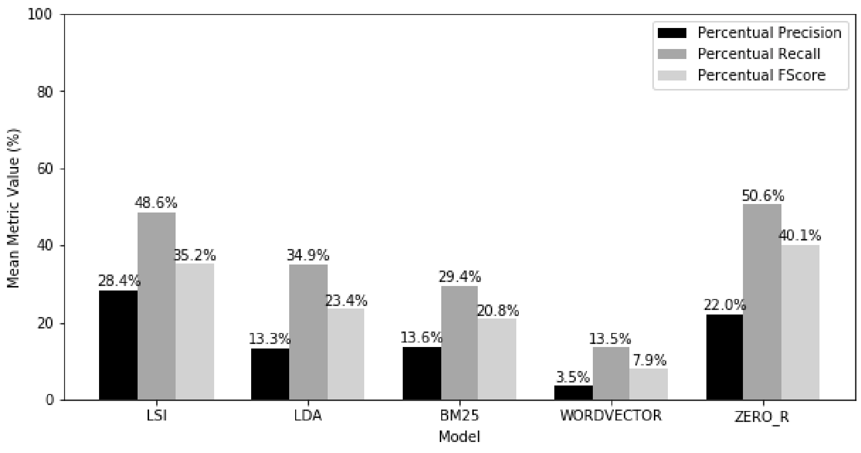


Figure 5.4: BR-TC Traceability Recovery Results

RQ1 – Which technique presents the best effectiveness?

In general, all selected IR and DL techniques presented poor results for all the metrics used. Analyzing the bar plot, we see LSI presented the best effectiveness in relation to the evaluated metrics ($Precision$, $Recall$, and F_2-$Score$). Surprisingly, LDA technique performed better than BM25 in terms of $Recall$ (34.9% for LDA and 29.4% for BM25) and F_2-$Score$ (23.4% for LDA and 20.8% for BM25). We expected the state-of-the-art IR technique (BM25) would achieve a better performance than the LDA. On the other hand, the Word Vector technique presented the poorest effectiveness in relation to all metrics with $Precision$ of only 3.5%, $Recall$ of 13.5% and F_2-$Score$ of 7.9%.

RQ2 – How does the effectiveness of each technique compare with a baseline predictor effectiveness?

Predicting only the *majority class*, our baseline predictor already has a $Precision$ of 22.0%. Also, we observe that only the LSI $Precision$ is bigger than the baseline's $Precision$, but for all the other techniques and metrics, the results are below the baseline's ones. Our baseline predictor was able to achieve a $Precision$ of 22.0% and a $Recall$ of 50.6% by indicating only positive links for the relationships between a bug report and the test cases with IDs *13,14,...,25* – corresponding to the *New Awesome Bar* system feature ones. In order to understand such poor results returned by all techniques, further investigations were carried out.

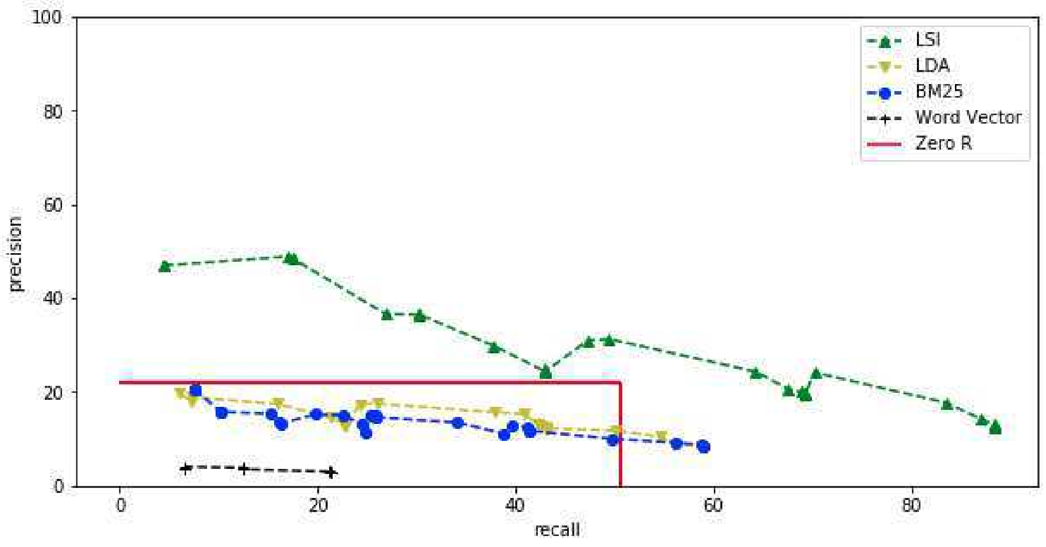
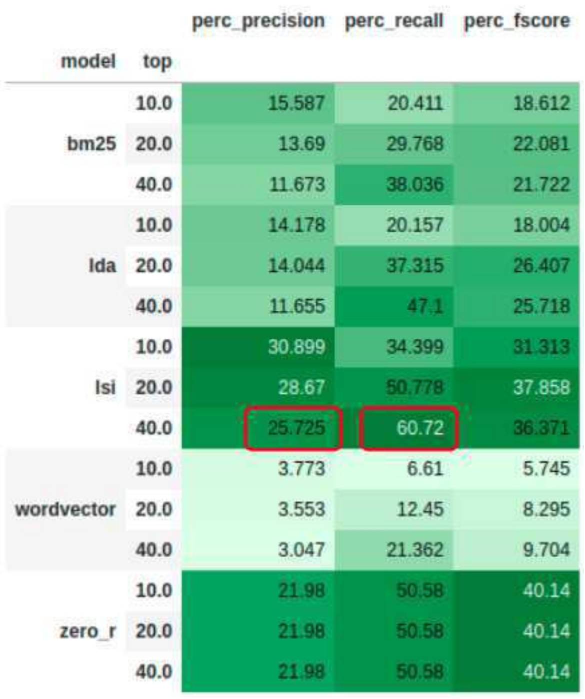


Figure 5.5: PR-Curves of All Techniques – BR-TC Context

Figure 5.5 details the PR-Curves of all techniques and the reference value of the ZeroR

Classifier's (in red) $Precision$ and $Recall$. The LSI's superior effectiveness in relation to the other technique is very clear: the values of $Precision$ and $Recall$ for every combination of Top Value and Similarity Threshold are always the highest for LSI and are always above the ZeroR scores.

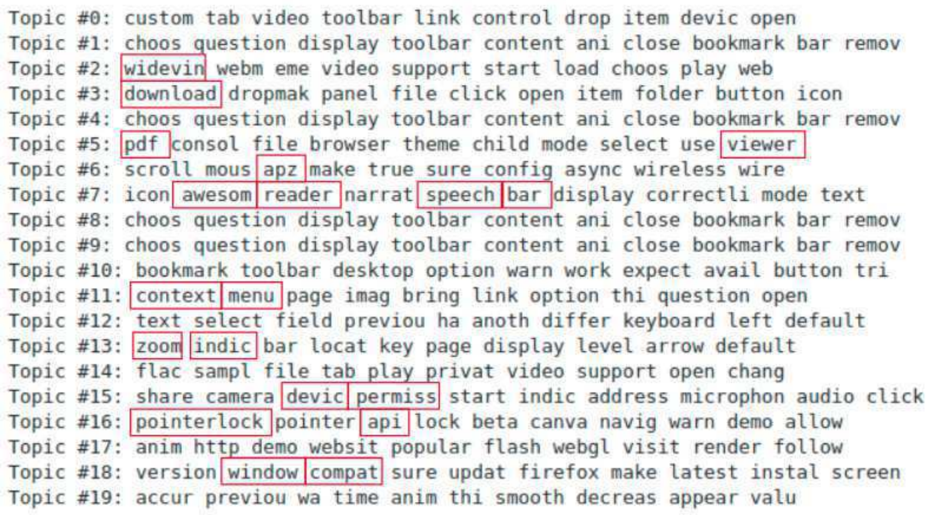
The LSI's effectiveness is confirmed if we look at the values aggregated by Top Value (10,20,40), as is shown in Figure 5.6. The darker green cells concentrate around the LSI technique for every Top Value and metric considered ($Precision$ percentage – **perc_precision**, $Recall$ percentage – **perc_recall**, and F_2-$Score$ percentage – **perc_fscore**), note the technique obtained higher $Precision$ and $Recall$ values for bigger Top Values, when compared with the ZeroR predictor – this indicates the fixed cut is influencing the techniques effectiveness. Although, we observed that for Top 40, the LSI obtained an *Acceptable* level of ***Goodness*** ($Precision$ > 20% and $Recall$ > 60%), what indicates its feasibility for traceability recovery tasks using such cut value in projects such as the Mozilla Firefox. Note the $Precision$ and $Recall$ highlighted in red at Figure 5.6.



model	top	perc_precision	perc_recall	perc_fscore
bm25	10.0	15.587	20.411	18.612
	20.0	13.69	29.768	22.081
	40.0	11.673	38.036	21.722
lda	10.0	14.178	20.157	18.004
	20.0	14.044	37.315	26.407
	40.0	11.655	47.1	25.718
lsi	10.0	30.899	34.399	31.313
	20.0	28.67	50.778	37.858
	40.0	25.725	60.72	36.371
wordvector	10.0	3.773	6.61	5.745
	20.0	3.553	12.45	8.295
	40.0	3.047	21.362	9.704
zero_r	10.0	21.98	50.58	40.14
	20.0	21.98	50.58	40.14
	40.0	21.98	50.58	40.14

Figure 5.6: Performance of techniques aggregated by Top Value

In the context of traceability of bug reports to test cases, the LDA technique was able to reproduce with much more trustworthiness the topics as system features, so the technique could split the test cases into groups which were very close to the features. Although, the technique was not able to achieve better results of $Precision$ and $Recall$ because low values of similarity that characterize some of these groups, and also due to some system features keywords that end up into the same topics. For example, the bug report 1357458, referent to the *New Awesome Bar* feature, was correctly related to the *New Awesome Bar* test cases, but also to the *Text to Speech in Reader Mode* test cases, because the tokens *awesom*, *reader*, *speech*, and *bar* all belong to the same topic in the technique's internal data structure. Observe the highlighted tokens in red for the 20 LDA topics in Figure 5.7.



```
Topic #0: custom tab video toolbar link control drop item devic open
Topic #1: choos question display toolbar content ani close bookmark bar remov
Topic #2: widevin webm eme video support start load choos play web
Topic #3: download dropmak panel file click open item folder button icon
Topic #4: choos question display toolbar content ani close bookmark bar remov
Topic #5: pdf consol file browser theme child mode select use viewer
Topic #6: scroll mous apz make true sure config async wireless wire
Topic #7: icon awesom reader narrat speech bar display correctli mode text
Topic #8: choos question display toolbar content ani close bookmark bar remov
Topic #9: choos question display toolbar content ani close bookmark bar remov
Topic #10: bookmark toolbar desktop option warn work expect avail button tri
Topic #11: context menu page imag bring link option thi question open
Topic #12: text select field previou ha anoth differ keyboard left default
Topic #13: zoom indic bar locat key page display level arrow default
Topic #14: flac sampl file tab play privat video support open chang
Topic #15: share camera devic permiss start indic address microphon audio click
Topic #16: pointerlock pointer api lock beta canva navig warn demo allow
Topic #17: anim http demo websit popular flash webgl visit render follow
Topic #18: version window compat sure updat firefox make latest instal screen
Topic #19: accur previou wa time anim thi smooth decreas appear valu
```

Figure 5.7: LDA Topics

The results of the Word Vector technique for that context of traceability were the lowest. Once again, the technique attributed high values of similarity for any pair of a test case and bug report, which a mean value of 0.91 and a standard deviation of 0.035. The implementation of the technique was not able to capture the nuances between documents and attribute different weights for the most relevant words in the test cases and the bug reports, so distinguishing relevant from not relevant test cases for a determined bug report. New strategies still need to be elaborated for this kind of technique.

In future work, we intend to explore variations of weighting schemes for specific targeted words in the vocabulary or make use of *enhancement strategies* [5] which better characterize the system features, so higher scoring values could be attributed to them. Also, strategies of preprocessing such as the applied by Merten *et al.* [31] could be replicated into our context of traceability (see Chapter 7).

In the next three sections, we extend our analysis and evaluate the results for two different scenarios and considering the range of Similarity Thresholds and Top Values, so we can answers the ***RQ3 – How does the effectiveness of each technique vary based on variable cuts?***. Due to the high number of combinations of Top Values and Similarity Thresholds, we selected two scenarios: (i) in the first scenario (Scenario I), the Similarity Threshold is 0.0, so the $Recall$ is favored over the $Precision$. (ii) in the second scenario (Scenario II), we evaluated the techniques using a Similarity Threshold of 0.9, such value leverages the $Precision$ over $Recall$ metric.

5.5.3 Scenario I – Similarity Threshold 0.0

The results for this first scenario (similarity threshold 0.0) are shown in Table 5.3 for each Top Value. Also, are depicted the number of traces captured by all techniques and the number of no captured traces.

Traces Missed by All Techniques We had a set of 36 no captured traces (7.0%), even when using the largest cut (Top Value 40). The missed traces for Top 40 are related to three system features: *Context Menu*, *Downloads Dropmaker*, and *New Awesome Bar*, where the majority is relative to the last one (28 bug reports). This phenomenon is also verified in the other Top Values, where the no captured traces are related with the *New Awesome Bar* feature in nearly 50% of the cases for Top 10 and 60% of the cases for Top 20. These results are coherent with the number of traces related to these features in the oracle, as detailed previously in Table 5.2, where more than half of the traces (260 out of 514) are linked to the *New Awesome Bar*, 93 to the *Context Menu*, and 72 to the *Downloads Dropmaker*.

Table 5.3: Captured and Not Captured Traces – All Techniques – Scenario I – Study II

Top	No Captured Traces	Traces Captured by All
10	$\frac{203}{514}$ = 39.49%	$\frac{6}{514}$ = 1.16%
20	$\frac{108}{514}$ = 21.01%	$\frac{27}{514}$ = 5.25%
40	$\frac{36}{514}$ = 7.00%	$\frac{70}{514}$ = 13.62%

We estimate the larger number of no captured traces is mainly due to the fixed cuts (Top Value) used: the number of no captured traces drops significantly with the increasing of the Top Value: no captured traces are only 7% in Top 40.

In order to better understand the missed traces, we analyzed the seven bug reports related to these missed traces:

- BR_1276720 (New Awesome Bar): there are no relevant keywords in the bug report content. The reporter used technical words or that do not belong to the test cases vocabulary, such as "searchbar" and "urlbar", which difficult the techniques task;
- BR_1279143 (New Awesome Bar): the description contains the word "awesomebar" written incorrectly. The presence of incorrect words requires the use of complementary techniques to detect and correct these mistakes before the query (bug report) be submitted to the technique, so a relevant result may be returned;
- BR_1296366 (New Awesome Bar): the bug description is very brief and the title contains the word "awesomebar", also written incorrectly, such as in the previous bug;
- BR_1293308 (New Awesome Bar): the bug reporter provided a technical description and used technical words, such as "urlbar", and a synonym "location bar", both not used in the test cases descriptions;
- BR_1270983 (Context Menu): this bug was probably reported automatically as result of automatic test failure. Despite the presence of the word "contextmenu" in the title, the technique was not able to link it with the test cases of this system feature. This may happen due to the writing, again *"incorrect"* into the bug description.
- BR_1299458 (Context Menu): this bug report is very well written, in fact citing the keyword "context menu" twice. We estimate the reason for not recovering the trace involving it is the slightly smaller size of the corresponding test case;
- BR_1432915 (Downloads Dropmaker): this bug report lacks important fields, such as the steps to reproduce and expected results. The reporter provided a very short description of a technical issue while downloading files. Despite the presence of the keyword "downloading", the techniques were not able to link this bug report with the test cases that have a shorter description and are relative to this system feature.

Traces Captured by All Techniques A small percentage of traces were recovered by all techniques for all Top Values, less than 15% were captured by all techniques, even for the largest cut (Top Value 40). However, some results can be highlighted: $\frac{27}{93} = 29.03\%$ of the traces linked with the features *Context Menu*, $\frac{12}{48} = 25\%$ of *Indicator for Device Permissions*, and $\frac{13}{72} = 18.05\%$ of *Downloads Dropmaker* were captured in Top 40.

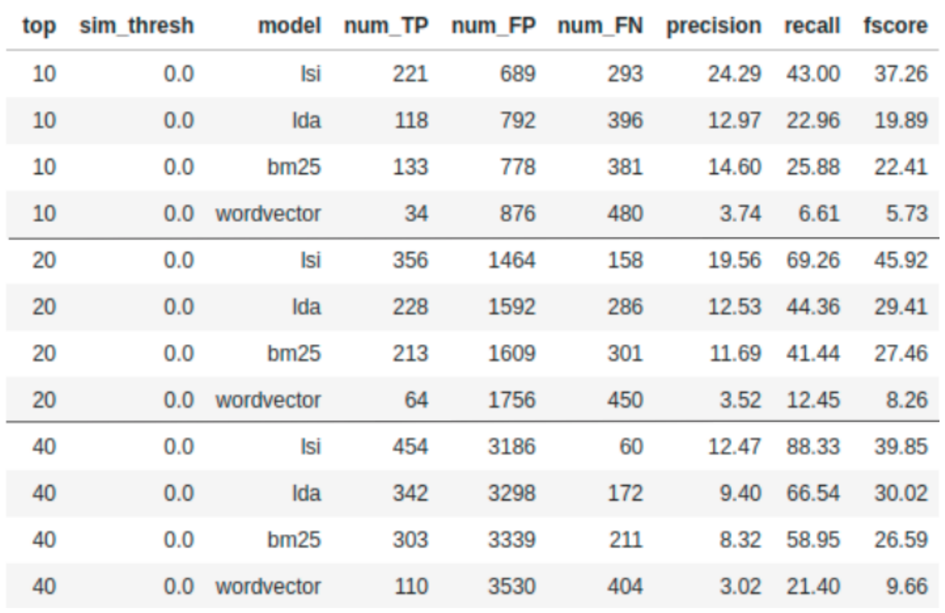
The test cases related to these system features, in general, are longer than the mean in terms of the number of words, which improves the similarity with truly related bug reports. Besides that, the test cases contain particular keywords that may highlight them to the traceability techniques. These words do not appear in other test cases because they are very linked to the context of these test cases – so there are less ambiguous usage of them into the studied context –, and also are often cited in the bug reports, so the developer can understand the context of the bug report before fixing it. Examples of such keywords are *"context menu"* for the homonym system feature; *"audio"*, *"video"*, and *"microphone"* for the system feature *Indicator for device permissions*; and *"download"* for the *Downloads Dropmaker* feature.

Techniques Evaluation

In this section, we evaluate each technique and explore some of the obtained results for each Top Value (10, 20, and 40), and the fixed Similarity Threshold 0.0. We detail the True Positives (TP), False Positives (FP), and False Negatives (FN) that characterize each technique. Figure 5.8 details the obtained results; note the increasing in the **recall** ($Recall$) values in parallel to the Top Values for all the techniques, while we have the decreasing of the **precision** ($Precision$) simultaneously. Whereas, if we consider the **fscore** metric (F_2-$Score$), we do not have a consensus about the best Top Value for all techniques.

True Positives (TP) The number of true positives of the LSI technique is considerably higher than the other techniques. In Top 40, the technique was able to recover nearly 88% of the relevant links ($Recall$ = 88.33%). Comparatively, the LSI correctly indicated 83 exclusive traces (traces that only it hit), while the LDA hit just 5, the BM25 hit 6, and the Word Vector hit only 3. The detailed results for each Top Value are shown in Figure 5.9.

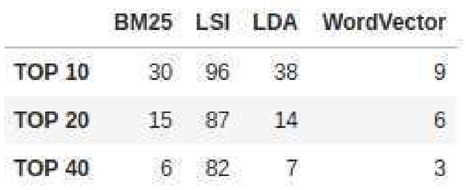
Through a qualitative analysis, we noticed the LSI technique was able to surpass common difficulties, such as the differences in the vocabulary used in test cases and bug reports, *i.e*



top	sim_thresh	model	num_TP	num_FP	num_FN	precision	recall	fscore
10	0.0	lsi	221	689	293	24.29	43.00	37.26
10	0.0	lda	118	792	396	12.97	22.96	19.89
10	0.0	bm25	133	778	381	14.60	25.88	22.41
10	0.0	wordvector	34	876	480	3.74	6.61	5.73
20	0.0	lsi	356	1464	158	19.56	69.26	45.92
20	0.0	lda	228	1592	286	12.53	44.36	29.41
20	0.0	bm25	213	1609	301	11.69	41.44	27.46
20	0.0	wordvector	64	1756	450	3.52	12.45	8.26
40	0.0	lsi	454	3186	60	12.47	88.33	39.85
40	0.0	lda	342	3298	172	9.40	66.54	30.02
40	0.0	bm25	303	3339	211	8.32	58.95	26.59
40	0.0	wordvector	110	3530	404	3.02	21.40	9.66

Figure 5.8: Traceability Recovery Results for Scenario I – Study II

the term "location bar" was used in some bug reports, but is not present in the test cases description, although the technique correctly linked the artifacts. Another difficulty is the presence of incorrect words, such as "awesomebar", which is referred to as "awesome bar" in the test cases.



	BM25	LSI	LDA	WordVector
TOP 10	30	96	38	9
TOP 20	15	87	14	6
TOP 40	6	82	7	3

Figure 5.9: Comparison of exclusive true positives

On the other hand, the other techniques hit a small number of exclusive true positive traces. However, they were able to hit some hard-to-trace links, for example, the LDA correctly identified a link between the bug report 1276120, which has two "incorrect" words ("urlbar" and "searchbar") and no other indication of the related system feature (New Awesome Bar).

False Positives (FP) In terms of false positives, the LSI technique is the best one for all Top Values, once it presented the smallest number. While the Word Vector had the largest number of FP also for all Top Values, such we can see in Figure 5.10. Although the behavior of all techniques is very similar numerically – the number of FP grows identically with the increasing of the Top Value –, we observed that the techniques incorrectly indicated traces relative to distinct system features. This suggests each technique has distinct preferences relative to the system features, although these are not directly used for doing the traceability – their text is not used by the techniques –, just the oracle used for the effectiveness evaluation.

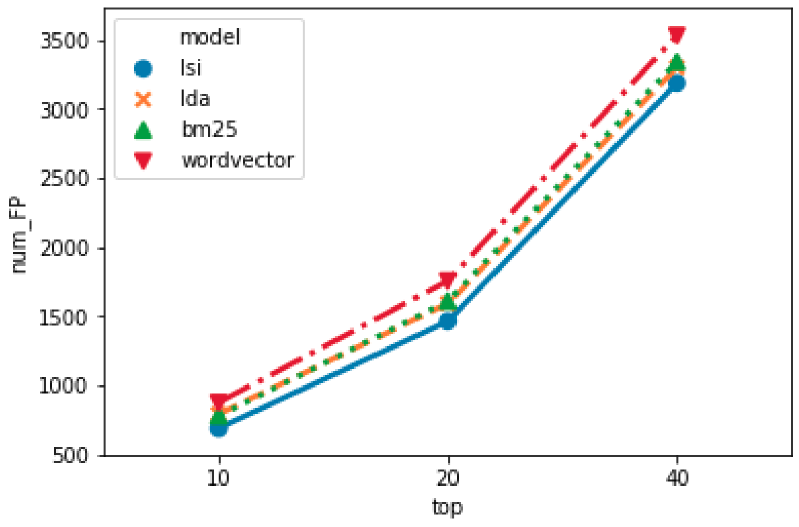


Figure 5.10: Comparison of false positives – Scenario 1 – Study II

The difference between the techniques' mistakes can be visualized looking at the heatmaps shown by Figures 5.11 and 5.12. Note that in the first heatmap (Figure 5.11) referent to Top Value 10, for example, the BM25 technique presented more FP related to the system feature *Download Dropmaker*, whereas the Word Vector presented more FP relative to *Context Menu* and *Pointer Lock API*.

An even more prominent behaviour is checked for higher Top Values, as we see in the second heatmap 5.12. In this case, the LDA makes more mistakes relative to the system features *Zoom Indicator*, *Text to Speech in Reader Mode*, *PDF Viewer*, and *Context Menu*. While the BM25 incorrectly indicated the presence of traces relative to *Downloads Dropmaker*, and the Word Vector to the *Context Menu*, *Pointer Lock API*, and *Windows Child Mode*. The most efficient technique identified (LSI) prefers the *New Awesome Bar* feature, which has the largest number of true traces associated to it (260 out of 514, see Table 5.2).

Next, we explore two examples of errors committed by the LSI at Top Value 40. The

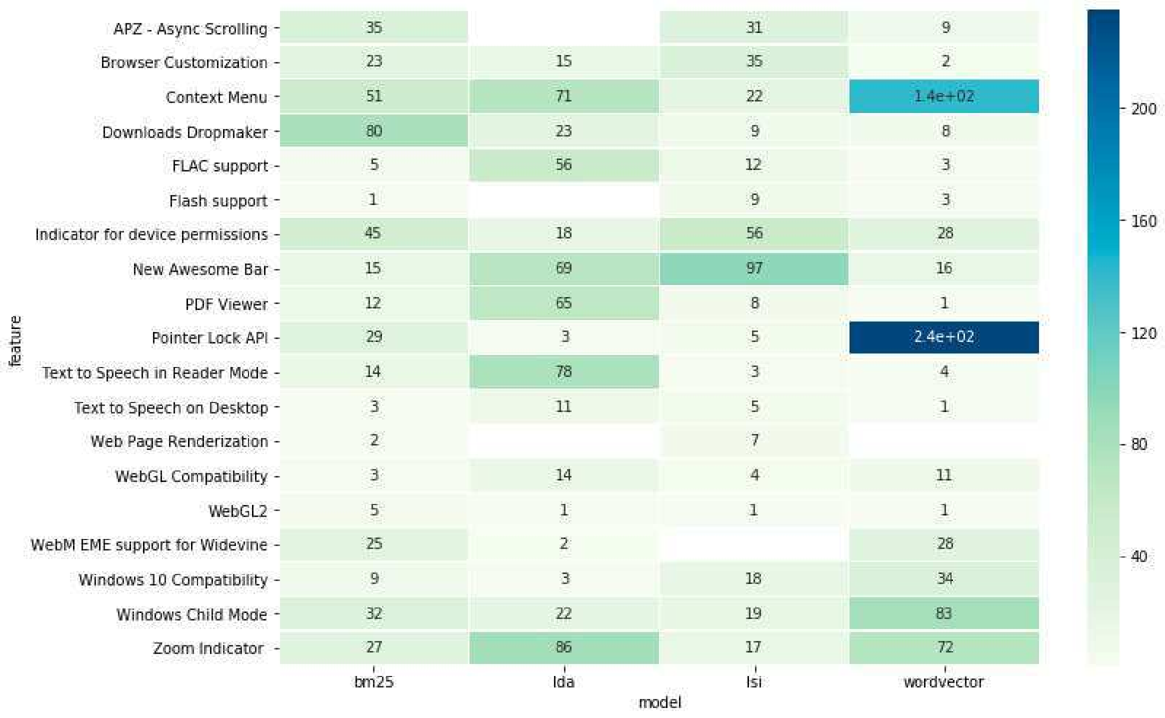


Figure 5.11: Comparison of number of false positives – Top Value 10

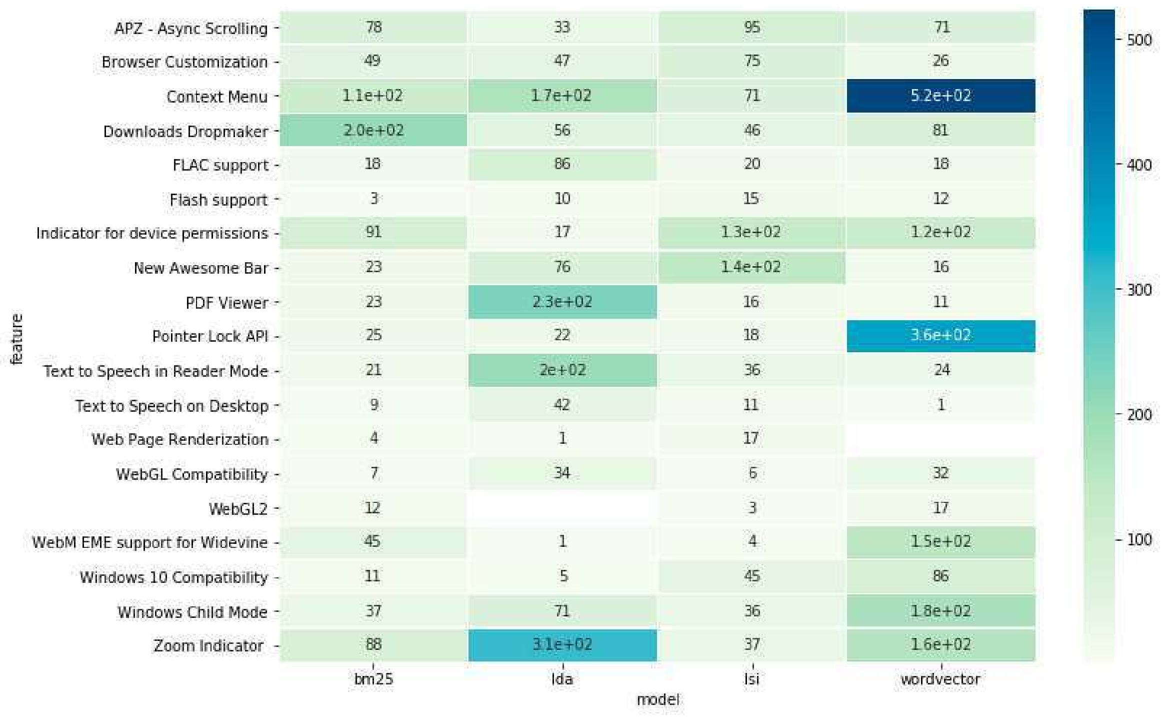
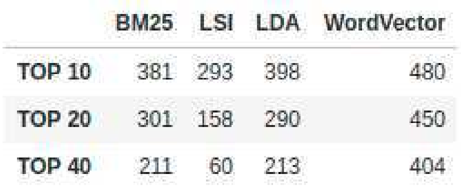


Figure 5.12: Comparison of number of false positives – Top Value 40

first one is related to the bug report 1269348, whose title is *"Show last sync date tooltip on Synced Tabs sidebar device names"*, the bug is not related to the *Indicator for device permissions*, however, the technique pointed out it as a positive link with nearly every test case from this system feature. Probably the technique was misguided by the presence of the word "device", understood differently in the test cases and bug report contexts. The second example is related to the bug report 1430603, which describes a technical issue involving source code implementation. In general, the description is very brief and technical, missing the recommended fields (steps to reproduce, expected results, etc.), we estimate the LSI indicated it as linked with test cases from eight different system features due to the large size of the cut (Top Value 40), otherwise these false positive traces would not exist.

False Negatives (FN) The Figure 5.13 shows the number of false negatives for each technique grouped by Top Value.



	BM25	LSI	LDA	WordVector
TOP 10	381	293	398	480
TOP 20	301	158	290	450
TOP 40	211	60	213	404

Figure 5.13: Number of False negatives – Scenario I – Study II

Although the numbers are elevated, we observed huge intersections between the sets of false negative traces, so that the LSI and BM25 techniques had no exclusive false negatives in Top 10. The details about these numbers are shown in Figure 5.14. How in the previously explained section about false positives, we have mostly disjoint sets of traces, in this case, traces not recovered by the techniques (false negatives). Observe the darker cells of each technique in Figure 5.14 and how they are distributed differently between the system features listed in the y-axis.

Investigating the exclusive false negative traces, we checked, in Top 40, the LDA made 13 out of 26 of its mistakes involving a single bug report, and eight out of 26 involving a single test case. The referred bug report (1299458), whose title is *"Telemetry data from Search bar is not properly collected when searching in new tab from context menu"*, originates all false negative traces related to the *Context Menu* feature (see Figure 5.14). An explanation for this may be the fact the bug report is also related to the *New Awesome Bar* feature – the issue mainly relates problems in recording the search bar telemetry data –, which may have misguided the LDA technique in recovering the traces.

Besides that, the technique presented difficulties into tracing links to the test case 14, whose title is *"Search State - Drop down"* and that belongs to the *New Awesome Bar* feature. We estimate the topics attributed to this test case were not enough to grant a minimum similarity score between each of the eight bug reports and this test case, so the links could be traced into the Top 40 cut. A probable cause for that may be the presence of more words in this test case description, it is longer than the other test cases associated with this system feature.

On the other hand, the Word Vector technique exclusively missed traces in relation to all the seven relevant system features (see Figure 5.14). However, the majority of missed traces is split between two features: *Downloads Dropmaker* (41) and *New Awesome Bar* (57). In the first case, just four bug reports are the source artifacts; whereas in the second case, are 12 bug reports. Some of these bug reports may be considered easy to trace, for example, the bug 1335992 (*"Search with default search engine stops working"*) which is correctly traced by the other techniques.

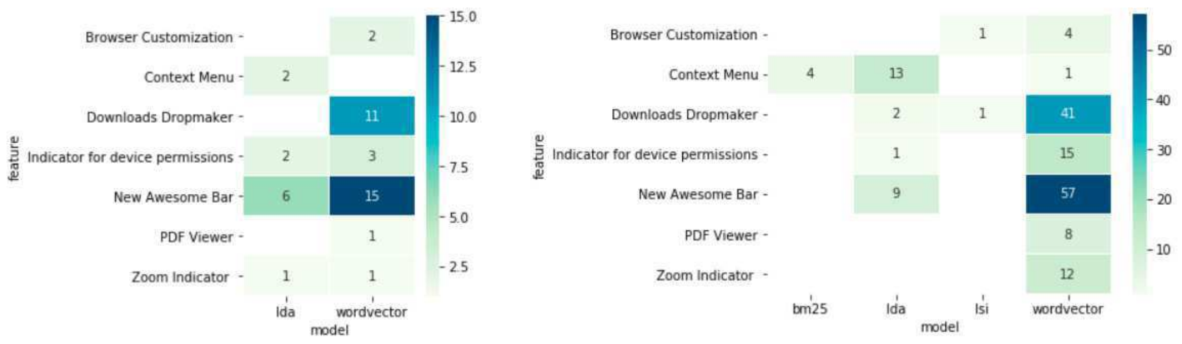


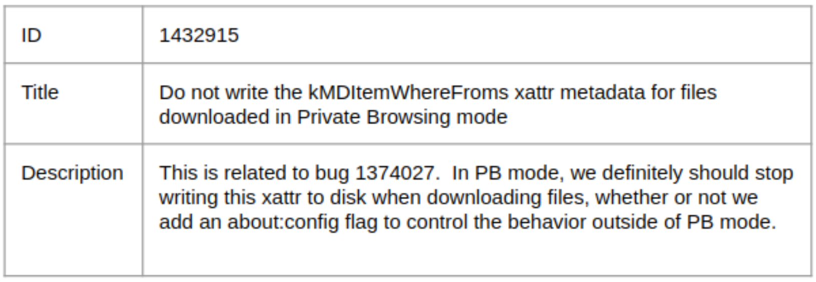
Figure 5.14: Comparison of exclusive false negatives – Top 10 (left) and Top 40 (right)

The Word Vector technique seems not to be able to distinguish the relevant and irrelevant artifacts, even for major cut values, such as Top 40. The algorithm adopted to calculate the similarity between two documents is very naive – once the mean of the vectors of the words in each document is calculated to then estimate the similarity between two documents (see Chapter 2) – and ignores the distinct weights the words may present. This may have caused the low performance of the Word Vector technique.

Whereas the BM25 – which uses a weighting scheme (*tf-idf*) for estimating the weight of each word considering the document it belongs and the entire set of documents in the corpus – achieved a better performance in terms of $Recall$ and had just four exclusive false negative traces in Top 40. All these traces were related to the *Context Menu* system feature and originated from only two bug reports.

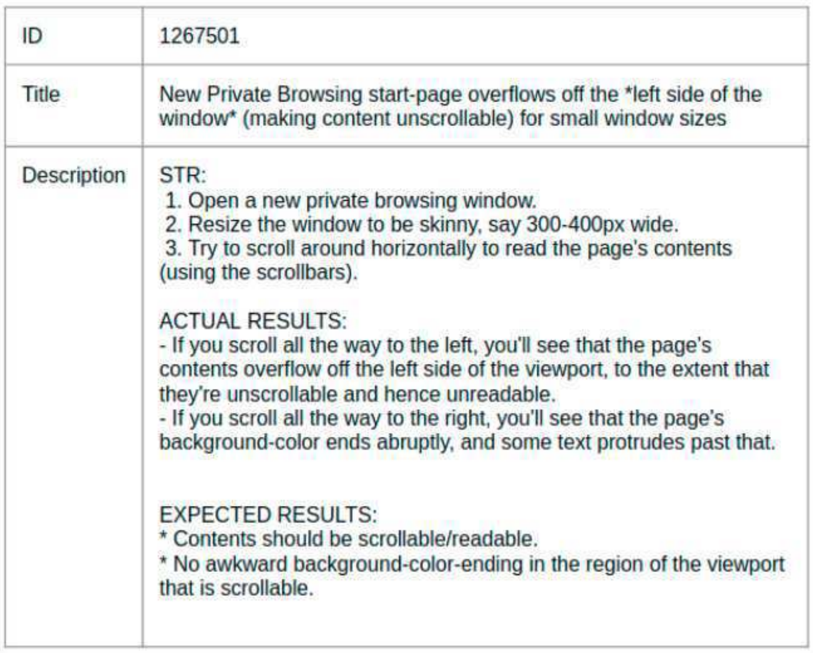
One of them is the same bug report (1299458) identified as the source artifact which generated all the exclusively missed traces of the LDA technique related to the *Context Menu* feature, however, related to another test case (92) in this case. This suggests this bug report may be especially hard to track. We estimate one reason for this difficulty is the fact it is related to two different system features simultaneously, although its focus in the *New Awesome Bar* feature. The same motive can be attributed for the second bug report (1248267), which originated the other three exclusive false negative traces, and also is related to the same system features.

When a bug report references more than one system feature, apparently is more difficult for the techniques to recover all the links. This problem is probably the cause the LSI was not able to recover one of the two traces (exclusive false negatives) it did not recover. The referred bug report (1357458) is related to *Browser Customization* and *New Awesome Bar* features, and its title is *"After Customization - typed text in the Awesome bar doesn't correspond with the text from One-Off-Searches bar"*. We noticed the words referencing the second feature are more often than the ones referencing the first feature, which may explain this behavior from the LSI technique. The other not recovered trace is relative to a poorly described bug report (1432915 – *"Do not write the kMDItemWhereFroms xattr metadata for files downloaded in Private Browsing mode"*) and the test case (162) from the system feature *Downloads Dropmaker*. Figure 5.15 shows bug report the 1432915 in detail – a poorly-described bug report –. Note how short is the description and how is hard to understand the reported issue. Whereas Figure 5.16 shows a well-described bug report. In this case, note the presence of the steps to reproduce, expected results, the actual results fields, and a title summarizing the bug report.



ID	1432915
Title	Do not write the kMDItemWhereFroms xattr metadata for files downloaded in Private Browsing mode
Description	This is related to bug 1374027. In PB mode, we definitely should stop writing this xattr to disk when downloading files, whether or not we add an about:config flag to control the behavior outside of PB mode.

Figure 5.15: Poorly described bug report



ID	1267501
Title	New Private Browsing start-page overflows off the *left side of the window* (making content unscrollable) for small window sizes
Description	STR: 1. Open a new private browsing window. 2. Resize the window to be skinny, say 300-400px wide. 3. Try to scroll around horizontally to read the page's contents (using the scrollbars). ACTUAL RESULTS: - If you scroll all the way to the left, you'll see that the page's contents overflow off the left side of the viewport, to the extent that they're unscrollable and hence unreadable. - If you scroll all the way to the right, you'll see that the page's background-color ends abruptly, and some text protrudes past that. EXPECTED RESULTS: * Contents should be scrollable/readable. * No awkward background-color-ending in the region of the viewport that is scrollable.

Figure 5.16: Well described bug report

5.5.4 Scenario II – Similarity Threshold 0.9

The same way we did in the first scenario, where we applied a similarity threshold of 0.0 (see the previous section), we evaluate each technique results in terms of True Positives (TP), False Positives (FP) and False Negatives (FN), so a detailed view of the obtained results is achieved. The considered Top Values were the same (10, 20, 40), but the fixed Similarity Threshold was 0.9. This threshold forces that a bug report and a test case have a high level of similarity to be traced a link between them. So, the $Recall$ scores are expected to be lower and the $Precision$ scores to be higher than in the first scenario.

Traces Missed by All Techniques A dropping in the $Recall$ values in practice mean there was an increase in the number of missed traces by the techniques. This is verified in Table 5.4, which presents the number of traces missed by all techniques and the number of traces captured by all techniques simultaneously considering the different Top Values.

Table 5.4: Captured and No Captured Traces – All Techniques – Scenario II – Study II

Top	**No Captured Traces**	**Traces Captured by All**
10	$\frac{413}{514}$ = 80.35%	$\frac{0}{514}$ = 0.0%
20	$\frac{382}{514}$ = 74.32%	$\frac{0}{514}$ = 0.0%
40	$\frac{344}{514}$ = 66.93%	$\frac{0}{514}$ = 0.0%

Through the analysis of the presented results, we see a larger number of no captured traces in this scenario and these numbers represents more than the double of the first scenario in Top 10 (39.49%), more than three times in Top 20 (21.01%), and more than nine times in Top 40 (7.00%). These results may be explained by the high similarity threshold (0.9) demanded to trace a link. Also, the fact the test cases have in general a small size, which means a small number of words, and these words were not enough to grant high levels of similarity between them and the bug reports.

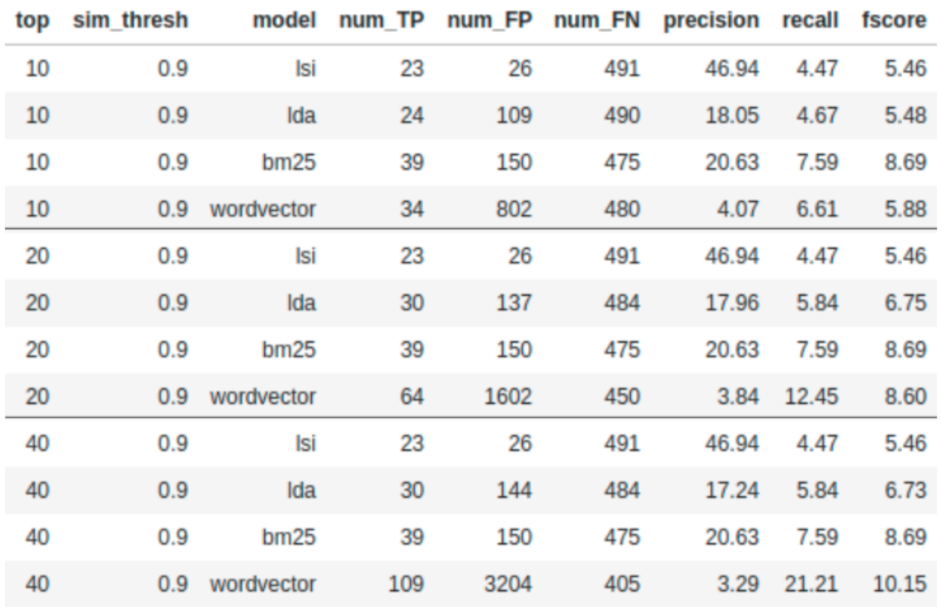
A primary conclusion we can make, considering these results, is that such value of Threshold Similarity is not feasible for the traceability recovery task between bug reports and test cases using the selected techniques. The majority of relevant links are not being recovered by any of the techniques.

Traces Captured by All Techniques Complementing the previous section, when we analyze the sets of traces captured by all techniques simultaneously, we observe that for any of the Top Values all the sets are empty, as we see in Table 5.4. Despite the fact only a small number of traces was captured by some technique (less than 35% in the best case – Top 40), this fact was a surprise.

Noting the difference between the results obtained in the first scenario and the second scenario, we raise the hypothesis we need variable similarity thresholds for the traceability recovery tasks between bug reports and test cases and they need to be adjusted for each technique individually. This hypothesis was already verified and experimented by other authors with works in the field [2; 9; 10], and our study corroborates their conclusions, despite the difference in the type of tracked artifacts. We analyze this hypothesis with more details in section 5.5.5.

Techniques Evaluation

In this section, we present and discuss the techniques true positives, false positives, and false negatives, which compose the $Precision$, $Recall$ and F_2-$Score$ metrics, such as in the first scenario. The traceability recovery results are detailed in Figure 5.17 with the highlighted Top Values. The number of true positives (num_TP), false negatives (num_FN), and false positives (num_FP) are detailed. How was expected the $Precision$ of the techniques was favored over the $Recall$ with this scenario, however, the $Recall$ values dropped significantly.



top	sim_thresh	model	num_TP	num_FP	num_FN	precision	recall	fscore
10	0.9	lsi	23	26	491	46.94	4.47	5.46
10	0.9	lda	24	109	490	18.05	4.67	5.48
10	0.9	bm25	39	150	475	20.63	7.59	8.69
10	0.9	wordvector	34	802	480	4.07	6.61	5.88
20	0.9	lsi	23	26	491	46.94	4.47	5.46
20	0.9	lda	30	137	484	17.96	5.84	6.75
20	0.9	bm25	39	150	475	20.63	7.59	8.69
20	0.9	wordvector	64	1602	450	3.84	12.45	8.60
40	0.9	lsi	23	26	491	46.94	4.47	5.46
40	0.9	lda	30	144	484	17.24	5.84	6.73
40	0.9	bm25	39	150	475	20.63	7.59	8.69
40	0.9	wordvector	109	3204	405	3.29	21.21	10.15

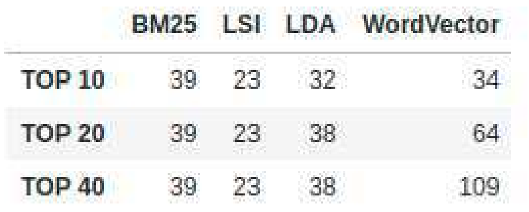
Figure 5.17: Traceability Recovery Results for Scenario II – Study II

All techniques presented very low $Recall$ scores, mostly below 10%, and these scores decay with the increasing of the Top Value. This is a critical issue, first because a high $Recall$ is a primary requirement for the usage of the techniques, so the majority of true traces are recovered and presented to the analyst/engineer – a high $Recall$ is more important then a high $Precision$; and second because it shows the techniques were not able to recover more traces, even if we double the Top Value at each cut (from 10 to 20, and from 20 to 40).

Whether we analyze the obtained values in relation to the $Goodness$ scale, we see none of the techniques presented satisfactory effectiveness in any of the evaluated Top Values. Also, the F_2-$Score$s are below the acceptable minimum under this scale ($Precision$ = 20% and $Recall$ = 60%, so F_2-$Score$ = 42.85%), all results are lower than 11%.

True Positives (TP) When we analyze the number of True Positives in Figure 5.17, we see the LSI technique had the lowest number of true positives, while the BM25 and Word Vector had the largest number. In order to better understand these results and highlight the differences between the techniques, we calculated the number of exclusive traces identified by each one of them. The results are depicted in Figure 5.19.

After analyzing it, we see that for Top 10 the techniques BM25, LDA and Word Vector have a close number of true positives, but the Word Vector was able to improve its performance with the increase in the Top Value, while the other techniques did not. We can see this phenomenon in Figure 5.18. The explanation for this phenomenon is the high similarity scores the technique attributed to nearly all the pairs of bug reports and test cases. How we mentioned at the beginning of the chapter, the mean value of the similarity scores is 0.907 and the standard deviation is very low (0.03), so is expected with the increasing of the Top Value that the number of true positives also increase.



	BM25	LSI	LDA	WordVector
TOP 10	39	23	32	34
TOP 20	39	23	38	64
TOP 40	39	23	38	109

Figure 5.18: Number of true positives by technique

As long as the Word Vector and LDA recover more trace links with the increasing of the Top Values, the number of exclusive traces recovered correctly by the other techniques decreases, such as we observe in Figure 5.19, where we split the traces by system feature (y-axis) and model (x-axis). Check how the LSI and BM25 "lose" traces for the LDA and Word Vector techniques. Note the darker cells, indicating a higher number of traces.

Also, we can check distinct system features being related to the recovered traces by each technique, indicating the existence of "preferences" between the techniques. For example,

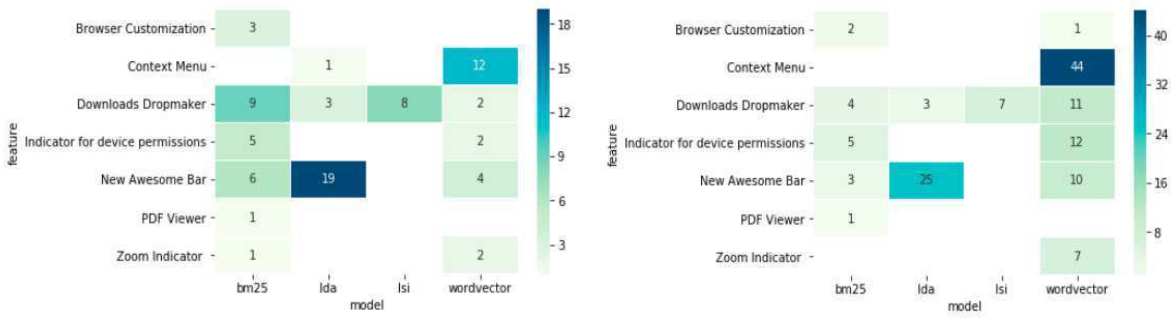


Figure 5.19: Comparison of exclusive true positives – Top 10 (left) and Top 40 (right)

in Top 10, the BM25 had nine exclusive traces related to the system feature *Downloads Dropmaker*, while the LDA had the majority (19) of traces linked to *New Awesome Bar*, and nearly half of the Word Vector traces are linked to the *Context Menu* feature.

These results enable us to see a possible complementarity between the techniques. We intend to explore this hypothesis in future works through the creation of a hybrid technique from the results obtained with these original four techniques. This hybrid technique in such scenario and with a Top Value of 10 would hit 78 traces out of the 514 possible.

False Positives (FP) Despite the LSI had the lowest number of true positives for all Top Values, it has the highest $Precision$ scores if compared with the other techniques. This is due to the low number of false positives presented by it, but not by the others. The number of false positives grows for every technique, except the LSI, which maintained the same 26 recovered traces, independent of Top Value, as we recognize in Figure 5.17. Observe that in the equation of the $Precision$, where the number of false positives is inversely proportional to the $Precision$ score (see Section 5.4.1).

The Word Vector technique is especially problematic due to its tendency to attribute high values of similarity between the test cases and bug reports even if they are not related. This tendency leads to the technique's high number of false positives, how larger the Top Value, larger the number of false positives. Remember the mean similarity value of Word Vector is around 0.907. Whereas the LDA and BM25 techniques did not suffer from the same problem and were able to limit the increment in their number of false positives.

An example of the referred Word Vector issue is the bug report 1248267, whose title is *"Right click on bookmark item of 'Recently Bookmarked' should show regular places context*

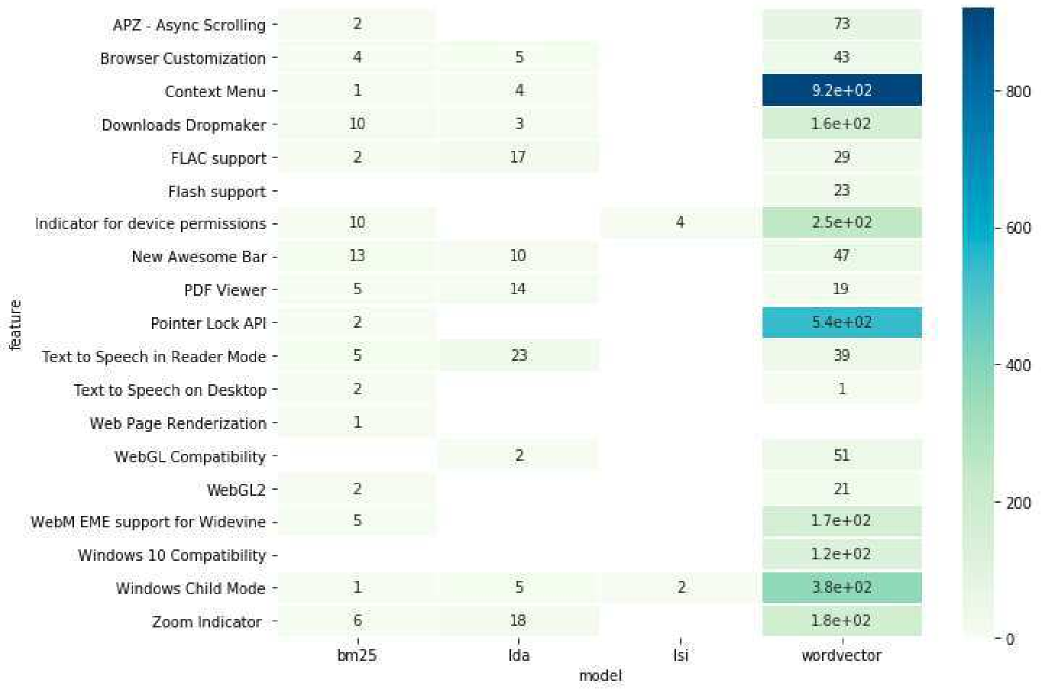


Figure 5.20: Comparison of exclusive false positives – Top 40 – Scenario II – Study II

menu" and is related with the *New Awesome Bar* and *Context Menu* system features, but the technique attributed high similarity scores (above 0.91) when comparing with test cases from these features and also with test cases from the *Windows Child Mode*. As already explained, this is mainly due to the lack of a weighting scheme for the words into the technique's algorithm.

Continuing our analysis of the false positives, we noticed differently from the false positives in the first scenario, the techniques did not distinguish themselves about the system features their traces are related to in this case. We can visualize this in Figure 5.20, which details the exclusive false positives of each technique (x-axis) and their respective system features (y-axis). Note the darker cells, indicating a large concentration of false positives traces, and how the larger values belong to the Word Vector technique.

False Negatives (FN) In what concerns the false negatives, which means the traces that were not recovered by the techniques but should, all of them had very poor results in this scenario. The $Recall$ values were below ten percent, except for the Word Vector in Top 20 and 40, such as we detect in Figure 5.17. Also, except for four exclusive false negative traces of LDA (in all Top Values), all of them had no exclusive false negative traces for any Top Value.

These results indicate the similarity threshold of 0.9 is not adequate for every technique, and an appropriate one must be determined for each one of them or a range of similarity thresholds must be used, as we did so that the effectiveness can be fairly calculated for each technique.

5.5.5 Best Similarity Threshold Value

In order to evaluate the hypothesis of existence of a best similarity threshold and to estimate it into the range of thresholds considered in this work, we conducted an analysis whose results are shown in Figures 5.21 and 5.22. The Figure 5.21 depicts the effects of the variation of the similarity threshold in the LSI and LDA techniques, while the Figure 5.22 shows the effects over the BM25 and Word Vector. We can visualize in each plot the $Precision$ (in blue), $Recall$ (in green), F_2-$Score$ (in brown), and the reference value for F_2-$Score$ (in red), so we can determine the level of *Goodness* (see Section 5.4.3). F_2-$Score$ values below this reference can not be considered *Acceptable*; the other levels of *Goodness* were omitted once none of the techniques achieved them and to not pollute the charts with excess of information.

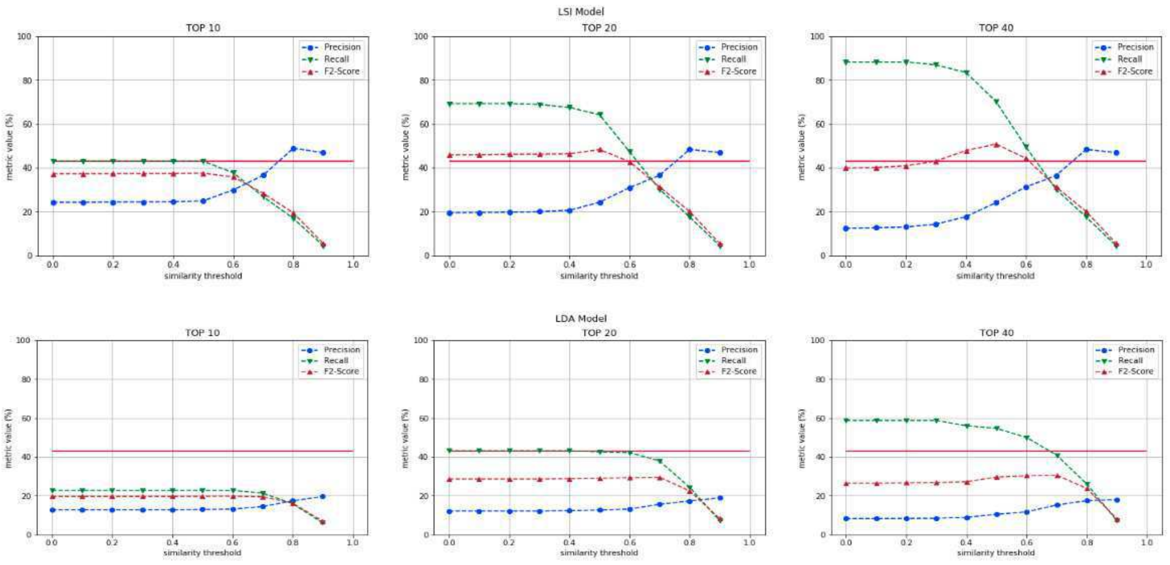


Figure 5.21: LSI and LDA Similarity Threshold Variation

RQ3 – How does the effectiveness of each technique vary based on variable cuts?

Analyzing both figures, we can visualize a clear difference between the behavior of $Precision$ and $Recall$ scores in the evaluated IR techniques and in the DL technique. In the first ones, the $Recall$ scores tend to fall below the $Precision$ scores beyond some similarity threshold independent of Top Value. For example, observe the turning point of the LDA technique for Top 10 near 20% for $Precision$ and $Recall$ and the similarity threshold of 0.8. Whereas the Word Vector technique practically suffer no influence from the similarity threshold, but from the Top Values and presented distinct, although constant, values of $Precision$ and $Recall$ for each Top Value (10,20,40) – note the straight lines in the Word Vector plots.

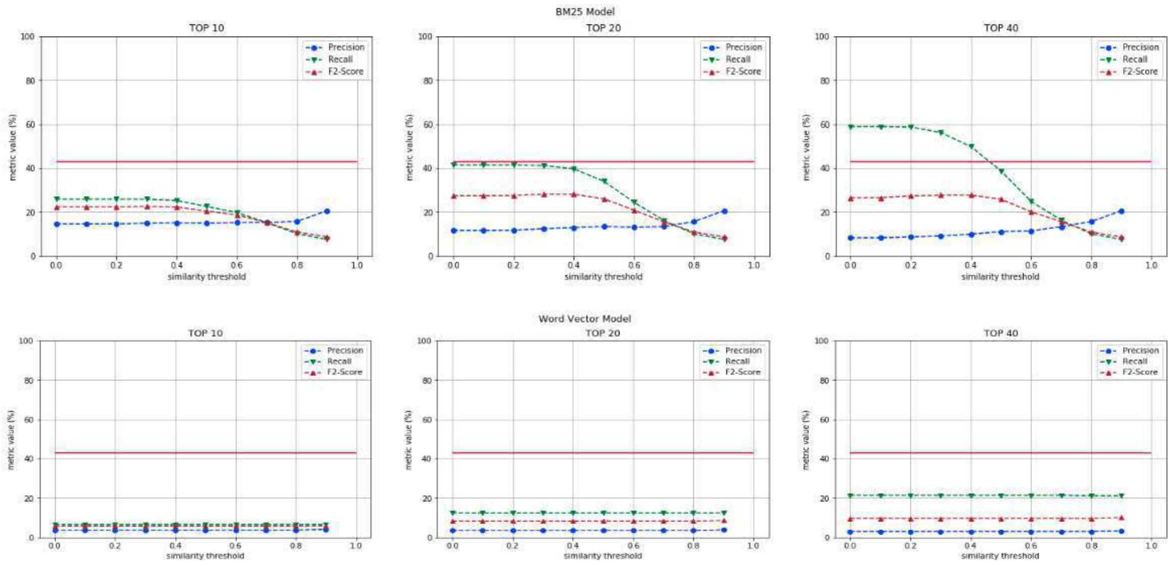
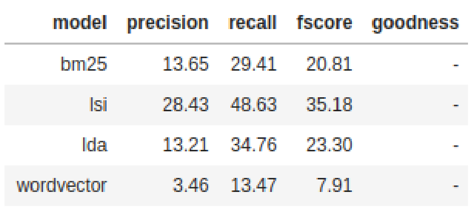


Figure 5.22: BM25 and Word Vector Similarity Threshold Variation

When we look at the F_2-$Score$ values, we see as expected the most of them is below the minimum value of reference (red line). This value split *Acceptable* techniques from the not satisfactory ones. The F_2-$Score$s of LDA, BM25, and Word Vector techniques are always below the reference value for every similarity threshold. However, the LSI technique presented some values which can be considered *Acceptable*: in Top 20, the similarity thresholds 0.0 to 0.5; and in Top 40, the similarity thresholds 0.4 to 0.6. In all these cut values the technique is *Acceptable*, which the highest level of acceptance for the combination Top 40 and Similarity Threshold 0.5 – this combination has a $Recall$ around 70% and $Precision$ near 23%.

5.5.6 Goodness Scale

Adopting the *Goodness* scale, we calculated the levels of acceptance of the $Precision$ and $Recall$ values for each technique and the results are shown in Figure 5.23. None of the studied techniques presented a satisfactory level of *Goodness* when we consider only the mean of $Precision$ and $Recall$ scores. Although, as explained in section 5.5.5, some combinations of Top Values and Similarity Thresholds grant an *Acceptable* level of *Goodness* for the LSI technique and one of them is identified as the most adequate one: Top Value 40 and Similarity Threshold 0.5.



model	precision	recall	fscore	goodness
bm25	13.65	29.41	20.81	-
lsi	28.43	48.63	35.18	-
lda	13.21	34.76	23.30	-
wordvector	3.46	13.47	7.91	-

Figure 5.23: Goodness Scale for each Technique – Study II

RQ4 – Which technique presents the best Goodness?

Such results indicate that LSI – using the identified best combination – is suitable for application in real and large projects as the Mozilla Firefox. The human analysts or engineers are able to recognize the correct and incorrect traces between a pair of a bug report and test case, as also to recover a considerable part of the trace links between these kinds of software artifacts when using a traceability recovery tool in their daily tasks.

5.5.7 Recovery Effort Index – *REI*

In this section, we report and analyze the *REI* values obtained for each technique considering all Top Values (10,20,40) and all Similarity Thresholds ([0.0, 0.1, ..., 0.9]). The $Precision$ score of the volunteers' oracle (produced only by the volunteers) in relation to the expert's oracle is 42.66%. This score is used to calculate the *REI* values (see Section 5.4.2).

RQ5 – Which technique presents the lowest REI coefficient?

We summarize the obtained *REI* values in Table 5.5. Since the *REI* coefficient is based on the $Precision$ scores and the LSI had the largest $Precision$ scores in this study, we expected it had the lowest *REI*, which in fact happened. The obtained results suggest the LSI

Table 5.5: *REI* values

Model	**REI**
BM25	2.06
LSI	0.90
LDA	2.19
Word Vector	11.51

is the less time-consuming technique – in relation to the time of analysis required from an analyst or engineer in using it for traceability recovery tasks – when compared with other techniques. The LDA and BM25 require nearly the double of LSI's required time, whereas the Word Vector nearly eleven times.

An important observation must be highlighted: we make a free association of *REI* values with the time required for analysis, such as did the authors of the original coefficient, but this association still needs deeper study and we cannot attribute statistical significance to it without further study.

5.5.8 Lessons Learned

We can highlight some conclusions from the developed study. First, the $Recall$ levels from three out of four techniques were below 40% – when we considered the average of the combinations of Top Values and Similarity Thresholds for each technique –, while $Precision$ levels remained below 30% for all techniques. In summary, when looking at the average values for the studied metrics, none of the techniques seems to have satisfactory levels.

One of the main reasons for that may be the terms used by bug reporters, which did not seem to match the terms used by Mozilla's QA team in test cases. This problem may be because most of the bug reporters do not participate from the Mozilla's testers teams, so the vocabulary used by the testers may not be present in most of the bug reports. The difference in vocabulary is a challenging problem that must be addressed and analyzed for the specific context of traceability between bug reports and test cases, especially in the case of real-world artifacts as the ones we work with. Specially in this problem, the use of system features would help the creation of a common vocabulary.

We also observed the quality of writing of the artifacts has a significant impact on the results achieved by the techniques. If a bug report, for example, is too short and do not describe precisely the problem, both human analyst and the technique will have difficulties in to recover the linked test cases. The establishment of guidelines to write a satisfactory bug report could be highly beneficial to the techniques effectiveness and to the engineers analyzing them.

Exceptionally in the Mozilla Firefox, the writing quality of test cases is high and the manual test cases are well-maintained by the QA team. For projects that do not count with high quality manual test cases, we suggest the adoption of approaches such as MBT (Model-Based Testing) [37], in which the test cases are generated automatically from a prefabricated model. The quality of the test cases generated is high and the maintenance of them is facilitated. Yet another suggestion is establishing guidelines for the manual production of test cases, where the fields required and the qualities of a satisfactory test case are highlighted.

Returning to the analysis of results achieved by the techniques, when we observe separately the different combinations of Top Values and Similarity Thresholds, we see the technique LSI presented a degree of *Goodness* for some of these combinations. The best-obtained result was for the combination Top Value 40 and Similarity Threshold 0.5, where the $Recall$ was nearly 70% and the $Precision$ nearly 23%.

The other three techniques had very poor effectiveness in this study, even looking at each combination of Top Value and Similarity Threshold separately. Although, we observed a possible complementarity between the techniques true positives, which suggests better effectiveness in using a hybrid technique that may be created based on the results presented by the studied techniques. We intend to explore that in future works.

Also, we observed there is still a considerable gap in the traceability recovery task for this type of traced artifacts. The results still not achieve a high level of *Goodness* (*Excellent* level) in relation to $Precision$, $Recall$, and $F_2\text{-}Score$, so that the recovery effort to recover the traces is the smallest possible from the involved people. The current level of *Goodness* grant an effective using of the LSI technique into a semi-automatized traceability recovery process, where we have the presence of human analysts or engineers working with the provided software tools for traceability recovery between bug reports and test cases, so that missing links can be traced and wrong links recovered by the tool can be filtered by the analyst.

The benchmark created in this work may help other authors in their studies in the traceability recovery field. The data set of bug reports, test cases, and system features, and the respective oracle matrix is the only available online to the best of our knowledge. The quality of the artifacts is preserved by the Mozilla's QA Team in the production of the test cases and system features, while the bug reports trustworthiness is attested by the applied filters in the proposed approach. Tangent to the oracle's building, we conducted a survey using a rigorous scientific methodology based on previous works. Other benchmarks were identified during the literature review phase of this work, although they do not use the same number or variety of traceability techniques, neither evaluate them through similar diversity of metrics. Also, it is important to highlight the very low number of studies found using bug reports and test cases, respectively as source and target artifacts, into the traceability recovery field.

We estimate the effort required from the analyst using the LSI is the smallest comparing with the other techniques and represents nearly half of the effort required when using the second best (BM25). However, this still needs further studies.

In terms of computational effort, we estimate the amount of resources required for using the Deep Learning techniques is bigger than the ones required by the Information Retrieval techniques. A significant amount of time was necessary to run the Word Vector techniques, when comparing with the Information Retrieval ones. We did not measure explicitly the respective times of execution of each technique, so further studies must be conducted to confirm or reject that hypothesis, this goes beyond the scope of this dissertation.

It is essential to highlight the importance of the task of traceability recovery between these two types of artifacts, especially in agile software development environments, where test cases are the most up-to-date documentation of the software, being fundamental for the software maintenance and evolution. Therefore, efforts should be continuous in order to reach the automatic and precise linking with the bug reports, thus increasing the robustness of software process and quality.

Chapter 6

Threats to Validity

In this chapter, we describe some threats to the validity of our study's conclusions.

One external threat is that the volunteers of the empirical studies do not participate from the Mozilla's testing and development teams so that they may classify some trace links incorrectly. However, this threat must be considered for deeper studies in the field, once errors in the creation of the oracle traceability matrix can exist even when it is created by developers and testers from the software project itself.

Similarly to the volunteers, the expert also had no participation in Mozilla's development and testing teams. Although, he had previous knowledge of information retrieval and deep learning techniques, and this could have caused some bias in his answers in the first empirical study. In order to eliminate this bias, we used the intersection of the volunteers' answers and the expert's ones, which implies in the need of agreement to accept an answer as right.

Another threat to our study is that we use only the Firefox artifacts to draw our conclusions. The single source of software artifacts limits the generalization of our conclusions which may be different when using other software systems. We intend in future works to extend the approach to other systems so that we can claim more generality for our conclusions.

Errors of implementation not detected in the script used in the empirical study for data processing and analysis is a threat to internal validity. However, we addressed this threat by double checking the produced software and eliminating existing programming errors previously to the analysis phase. Additionally, we open-sourced our code which is available online.

Due to recording failures in the application used for the empirical study, two out of the

93 tasks needed to be discarded. Therefore, two bug reports were also discarded. We believe this represents a minor threat to our conclusions and does not impose a significant risk to it given the amount of remaining tasks/bug reports with correct answers.

Chapter 7

Related Work

Comparisons between techniques in the traceability recovery context were carried out in previous studies. Falessi *et al.* [14] characterize and compare different IR techniques, with distinct parameters, for equivalent/redundant requirements identification. The focus is on requirements documents for an industrial system, in which five evaluation metrics (Precision, Recall, ROC area, Lag, and Credibility) were employed. They analyze algebraic models and vary term extraction strategies, weighting schemes, and similarity metrics (Cosine, Dice, and Jansen-Shannon); by testing many combinations of these variables, they propose the most efficient for the metrics they selected. Our proposal evaluates a larger amount of IR and DL techniques, not only algebraic techniques comparing their effectiveness in terms of $Precision$, $Recall$ and F_2-$Score$, providing a broader perspective over the technique's differences.

Similarly, Mills [33] applies a set of popular machine learning models/techniques – except Neural Networks, different from our study – for classifying possible trace links as positive (1) or negative (0), for a pair of textual software artifacts, which did not include bug reports to test cases. An extensive set of variables, extracted from historical data about the traced artifacts, was used for the training of the models/techniques, and a comparison between them is drawn in terms of $Recall$ and *False Positive Rate* (FPR). The author uses several artifacts such as use cases, test cases, and source code, but not bug reports.

Regarding bug reports and test cases, Kaushik *et al.* [25] study traceability recovery for a private industrial system using $Precision$, $Recall$, and F_1-$Score$ metrics. However, they select LSI and LDA as IR techniques – they did not use BM25 and DL techniques – and set up a constant similarity threshold of 0.7 for trace links, and a range of top values (2,5,10) – our study was performed in more diversified settings and with a larger amount of bug reports. In their study, they have access to a tester who created the oracle, while ours was built with the aid of volunteers, as a superset of all answers. This difference grants them greater oracle reliability when compared to our approach, although it is not necessarily better, once relies only upon one person's answers. Also, they discuss two scenarios for linking test cases to bug reports: one considering the test case's *folder name* (as we did with system features) and another considering only the direct match between the recovered traces and the oracle traces. In their results, LSI performs better than LDA, corroborating with our results. Concluding their work, the authors observed the better effectiveness of LSI over the LDA as we did, especially for the first scenario (using folder's names). We can not directly compare our results with theirs, once their conclusions were expressed only in terms of F_1-$Score$, while ours do not calculate this metric, but F_2-$Score$.

Merten *et al.* [31] analyze a set of five IR techniques for recovering of traceability links from bug reports to bug reports in four different opensource projects. The selected techniques were VSM, LSI, BM25, BM25+ and BM25L with and without the application of preprocessing steps (stop words removal, stemming, etc.), and also evaluating different weighting values attributed for distinct parts of the bug report, for example, *title*, *source code*, *stack trace*, *comments*, etc.. The authors pursued similar metrics to ours: $Precision$, $Recall$, and the *Goodness* scale. Besides these metrics, they also compared two versions of F-$Score$: a balanced version (F_1-$Score$) and an unbalanced version (F_2-$Score$), which gives more importance to $Recall$ over $Precision$. The baseline for comparison between the techniques adopted by them was the BM25 technique, while we decided to use a ZeroR classifier.

The conducted study by Merten *et al.* verifies the superior effectiveness of the LSI technique over the BM25 when involving bug reports textual analysis, however, all techniques perform poorly just as in our study. Although we make traceability between different types of artifacts, we may observe similar results, given the similar nature of the query artifacts (bug reports).

Merten *et al.* also highlights the difficulties into track bug reports, such as the presence of *noise* in the bug report text, such as hyperlinks, source code, stack traces, and repetitive information. The presence of such *noise* affects the effectiveness of the techniques and must be addressed as we also have identified in our study.

Table 7.1 shows a summary of the related works, the used techniques, the artifacts mapped and the similarity metrics used.

Table 7.1: Summary of related works

Work	**Technique**	**Artifacts**	**Sim. Metric**
[14]	LSI,VSM	RD[a]	Cosine,Dice,Jansen-Shannon
[33]	J48,KNN,NB[b],RF[c]	UC[d],SC[e],RD[f],TC[g],ID[h]	–
[25]	LSI,LDA	TC,BR[i]	Cosine
[31]	LSI,BM25	BR	BM25,Cosine
This work	LSI,LDA,BM25,WV[j]	BR,TC	Cosine

[a]RD: Requirement documents
[b]NB: Naive Bayes;
[c]RF: Random Forest
[d]UC: Use Cases
[e]SC: Source Code;
[f]RD: Requirement Documents;
[g]TC: Test Cases;
[h]ID: Interaction Diagrams;
[i]BR: Bug Reports
[j]WV: Word Vector

Chapter 8

Conclusions

In this dissertation, we propose an approach to recover traceability links between bug reports and test cases, through the use of system features to bridge the gap between those two types of artifacts. Several IR and DL techniques may be used as instantiations of the approach. We compared the effectiveness of these techniques and stated the better effectiveness of a traditional technique (LSI) in terms of well-known metrics in the context of traceability from bug reports to test cases over the other studied techniques. In special, we have also addressed the applicability of one DL technique – in this case, Word Vector – for traceability recovery, which presented the poorest results.

The comparative analysis made in this work, involving such number of distinct traceability techniques from different families and recovering traceability links between test cases and bug reports using system features as intermediate artifacts states the originality of this work. Through the use of well-known metrics and adapted ones a broader overview of the techniques effectiveness was drawn when comparing with previous studies in the field. Also, the case study conducted with Mozilla Firefox real artifacts strengthen the achieved results when replicating the approach in other large and open-source projects.

Although the results may suggest the using of the available IR and DL techniques for automatic traceability recovery, we checked that, in real and large software projects such the Mozilla Firefox, it is still unfeasible for complete automation. Our proposal and studies reveal the strengths and weaknesses of each applied technique and identified the feasibility of the LSI technique using some combinations of Similarity Thresholds and Top Values. Once we set up the LSI technique with these combinations – preferably the best one –, into an appropriate tool, then it may aid human analysts and engineers in semi-automatized traceability recovery tasks.

Even with the stated feasibility of one of the techniques, we checked that the presence of a common vocabulary and the proposal of a guide for writing the bug reports and the test cases can greatly benefit the process of traceability recovery involving these two kinds of artifacts. Besides that, we have identified the possibility of using the system features as a link between bug reports and test cases, which *per se* is a contribution of this research, and whose adoption for describing bug reports and test cases potentially improves the traceability effectiveness of the traceability recovery techniques.

The results achieved in this work can be replicated using the script and data available online. Also, the data set of extracted bug reports, test cases, system features, and the created oracle may serve as a benchmark for other studies in the traceability recovery field. During the literature review phase of this work, we noticed the existence of other benchmarks which are also available online, although they do not map bug reports to test cases. Therefore, the one created is the first of this kind from the best of our knowledge. Also, the quality of data used is improved by the fact the bug reports are confirmed and fixed, the test cases were produced directly by the qualified QA team of Mozilla, the system features descriptions were collected from trusted sources, and the oracle creation process follows an well-described scientific methodology.

The proposed approach has the potential to be adopted in various scenarios in a software development process. For instance, it could be used to aid human analysts to evaluate the impact of changes and to help testers to select and prioritize manual test cases related to a determined bug report. Currently, the only requirements for using our approach is to provide manual test cases, bug reports and system features in a textual format, and that the bug reports and manual test cases be grouped by system features.

We emphasize the test cases we use in the proposed approach are manual, not automatized. In the case of automatized test cases, such as unitary tests and integration tests, which are defined through the use of programming languages, we estimate the results would be even worse, once the technique would have more difficulties into finding the most relevant terms and, therefore, recovering the correct test cases linked with a given bug report.

8.1 Limitations

The presented work has some limitations that should be highlighted. In specific, the main constraints regard the oracle production, the technique's parameters selected, and the statistical significance of the results.

Concerning the oracle production, we tried to minimize the errors of generation by taking the intersection between the answers of volunteers and an expert, but a more robust generation process could have been designed, so that a bigger amount of traces could be used to create the oracle, and less information (traces) would be discarded during this process of creation. Remember we only used the traces presented in the intersection of the volunteers' and the expert's answers.

Additionally, a deeper parameter searching process could have been made, so the techniques used would adopt the most adequate parameters for the software artifacts in our data set. This process was carried out without further indications that the chosen parameters were really the best ones.

Besides that, the statistical significance would give more robustness to the choice of the parameters, and also to the studies results, which would leverage the effectiveness comparison between the various techniques. However, statistical tests were not used for consolidating the results obtained, so we have no indication of the difference between the techniques from a statistical standpoint. The use of confidence intervals, for example, could have suggested a higher similarity between the effectiveness of the techniques.

8.2 Contributions

In short, the contributions of this work are:

1. Organization of a data set of bug reports, test cases, and system features from Mozilla Firefox for using in traceability recovery studies and support the creation of new benchmarks of traceability techniques;

2. Execution of a comparative study of the IR and DL techniques used in terms of effectiveness;

3. An approach for traceability recovery between bug reports and test cases using system features;

4. An extra study was conducted (see Appendix A) for evaluating the effectiveness of each IR and DL technique between *bug reports and system features*.

8.3 Future Work

As future work, we have some paths we can follow. One is to similarly compare with other techniques, in particular, DL techniques using neural networks trained with software engineering domain data sets, extending the analysis for systems from both the open-source community and private sector.

Another path is the application of "enhancements" strategies with the LSI technique, such as building a thesaurus to deal with synonym, clustering of documents/terms, phrasing, query expansion techniques, and vary the attributed term weight considering the localization of them, for example, when a term appear in the title, it gains a higher weight than if it appeared in the comments of the bug report.

Yet another possible path is to evaluate the effectiveness of a hybrid technique created from the answers (returned traces) of the studied techniques. Hopefully, the number of mistakes (false positives and false negatives) may be diminished if compared with the individual effectiveness of each technique.

How we checked during our study, the techniques tend to hit and miss different sets of traces, so that we suppose a combined version of them – in the form of a hybrid technique – can compensate the failures of each technique individually, and also boost the number of correct traces recovered.

Bibliography

[1] Tf-idf: A single-page tutorial - information retrieval and text mining. Retrieved May 30, 2019 from http://www.tfidf.com/.

[2] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[3] Daniel M. Berry. Evaluation of tools for hairy requirements and software engineering tasks. *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference Workshops, REW 2017*, pages 284–291, 2017.

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, (3):993–1022, 2003.

[5] Markus Borg, Per Runeson, and Anders Ardö. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering*, 19(6):1565–1616, 2014.

[6] Stefan Buttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.

[7] Gerardo Canfora and Luigi Cerulo. Fine Grained Indexing of Software Repositories to Support Impact Analysis. *Advanced Materials Research*, sep 2006.

[8] Steven Davies and Marc Roper. What's in a bug report? *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*, pages 1–10, 2014.

[9] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Can information retrieval techniques effectively support traceability link recovery? *IEEE International Conference on Program Comprehension*, 2006:307–316, 2006.

[10] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 14(1):57–92, 2009.

[11] Scott Deerwester, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by Latent Semantic Analysis. *J. Am. Soc. Information Science*, 41(6):391–407, 1990.

[12] Alex Dekhtyar and Vivian Fong. RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow. *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference, RE 2017*, pages 484–489, 2017.

[13] Alex Dekhtyar, Jane Huffman Hayes, Senthil Sundaram, Ashlee Holbrook, and Olga Dekhtyar. Technique integration for requirements assessment. *Proceedings - 15th IEEE International Requirements Engineering Conference, RE 2007*, pages 141–152, 2007.

[14] Davide Falessi, Giovanni Cantone, and Gerardo Canfora. A comprehensive characterization of NLP techniques for identifying equivalent requirements. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, page 1, 2010.

[15] Davide Falessi, Massimiliano Di Penta, Gerardo Canfora, and Giovanni Cantone. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering*, 22(3):996–1027, 2017.

[16] Mattia Fazzini, Martin Prammer, Marcelo D'Amorim, and Alessandro Orso. Automatically translating bug reports into test cases for mobile apps. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, pages 141–152, 2018.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2016.

[18] Orlena C Z Gotel and Anthony C W Finkelstein. An Analysis of the Requirements Traceability Problem. *1st International Conference on Requirements Engineering (RE 1994)*, pages 94–101, 1994.

[19] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically Enhanced Software Traceability Using Deep Learning Techniques. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 3–14, 2017.

[20] Jane Huffman Hayes and Alex Dekhtyar. Humans in the traceability loop: can't live with'em, can't live without'em. *3rd international workshop on Traceability*, 2005.

[21] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Tracing and Mapping : Supporting Software Quality Predictions. 2005.

[22] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[23] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, E. Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. REquirements TRacing On target (RETRO): Improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, 2007.

[24] Hadi Hemmati and Fatemeh Sharifi. Investigating NLP-Based Approaches for Predicting Manual Test Case Failure. *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pages 309–319, 2018.

[25] Nilam Kaushik, Ladan Tahvildari, and Mark Moore. Reconstructing traceability between bugs and test cases: An experimental study. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 411–414, 2011.

[26] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. pages 31–40, 2005.

[27] Dennis Lee. How to write a bug report that will make your engineers love you, 2016. Retrieved May 30, 2019 from https://testlio.com/blog/the-ideal-bug-report.

[28] Marco Lormans and Arie Van Deursen. Can LSI help reconstructing requirements traceability in design and test? *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pages 47–56, 2006.

[29] Mika V. Mäntylä, Foutse Khomh, Bram Adams, Emelie Engström, and Kai Petersen. On rapid releases and software testing. *IEEE International Conference on Software Maintenance, ICSM*, pages 20–29, 2013.

[30] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–82, 2012.

[31] Thorsten Merten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech. Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9619, pages 45–62. 2016.

[32] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *CrossRef Listing of Deleted DOIs*, 1:1–12, jan 2013.

[33] Chris Mills. Automating traceability link recovery through classification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 1068–1070, New York, New York, USA, 2017. ACM Press.

[34] Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, Andrea De Lucia, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. *IEEE International Conference on Program Comprehension*, pages 68–71, 2010.

[35] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented software evolution. page 1, 2013.

[36] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods*

in Natural Language Processing (EMNLP), pages 1532–1543, Stroudsburg, PA, USA, apr 2014. Association for Computational Linguistics.

[37] Alexander Pretschner. Model-Based Testing in Practice. *International Conference on Software Engineering, 1999. Proceedings.*, pages 537–541, 1999.

[38] Stephen Robertson and Hugo Zaragoza. *The Probabilistic Relevance Framework: BM25 and Beyond*, volume 3. 2009.

[39] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2010.

[40] Ian Witten, Eibe Frank, and Mark Hall. *Data Mining - Practical Machine Learning Tools and Techniques*, volume 54. 2011.

[41] Andrew Y. Ng and Michael Jordan. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. *Adv. Neural Inf. Process. Sys*, 2, 2002.

[42] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: An application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, 2005.

[43] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, Cathrin Weiss, Adrian Schröter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, sep 2010.

Appendix A

Extra Empirical Study A

In this Appendix, we present an extra study conducted by us in order to analyze the techniques in the context of traceability between bug reports and system features.

A.1 Study Context and Definition

The objective of this study is to analyze the traceability recovery capabilities of the IR and DL techniques used in this work with the created oracle between bug reports and features of the Mozilla Firefox (see Chapter 4). We aim to answer the following research question: *RQ-1A: Which technique presents the best effectiveness in the context of traceability between bug reports and system features?*. We define effectiveness in terms of common metrics used in the field: $Precision$, $Recall$, and F_2-$Score$ [22; 3].

A.2 Study Procedure

The execution procedure of this study is identical to the one presented in the Second Empirical Study (see Chapter 5) regarding the execution of techniques and trace links matrices recovery. The only difference is the targeted artifacts since we evaluate in this study the mapping between system features and bug reports, so the oracle transformation is not necessary, we have access to the oracle between bug reports and system features provided by the participants in the First Empirical Study (see Chapter 4). Figure A.1 shows a schematization of this study in detail.

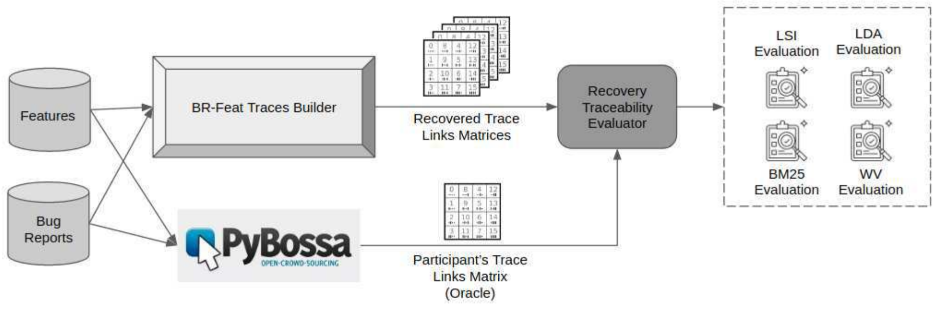


Figure A.1: Scheme of Extra Empirical Study

As the input of the scheme, we see two data sets of System Features and Bug Reports entering the *BR-Feat Traces Builder* and the *PyBossa Platform*. The first, as explained in Chapter 3, produces binary matrices for each combination of Top Value and Similarity Threshold for each technique. Whereas the second provides the oracle generated through the volunteers' and expert's participation, as was already explained. Then, the module *Recovery Traceability Evaluation* receives both outputs and evaluate the effectiveness of the techniques.

A.2.1 BR-Feat Traces Builder Evaluation

The evaluation of the *BR-Feat Traces Builder* occurs identically to the described in Chapter 5 for the *BR-TC Traces Builder*. The *Traceability Engine* also processes the input artifacts in two phases: preprocessing and execution. The preprocessing steps and parameters set up are exactly the same, such as the libraries and frameworks.

In order to find the best parameters values for the LSI (number of dimensions) and the LDA (number of topics), a parameter search was performed. We tested the implemented techniques with smaller and greater parameter values (5,10,20,40,100), but coincidentally 20 was the best value for both LSI and LDA techniques. Whereas the BM25 and Word Vector implementation and parameters were identical to the ones used in the Second Empirical Study.

Following the preprocessing phase the techniques were executed with the tokenized bug reports and features, similarity matrices (see Chapter 2) were then generated, and different *BR-Feat Recovered Trace Links Matrices* were created according to multiple combinations of *Top Values* and *Similarity Thresholds*. For Top Values, we used 1, 3, and 5 – which is compatible with the average number of features to be recovered by bug report, and a range of similarity threshold values between 0.0 and 0.9 (included) with a step size of 0.1 (0.0, 0.1, ..., 0.9) – which is compatible with the range of the values into the similarity matrices that interest us (the ones with similarity greater than zero, meaning closest similarity between the documents). Finally, the *Recovered Traceability Evaluator* assessed each technique using selected metrics and the participant's trace links matrices (oracles).

A.3 Research Method

The same evaluation procedure adopted for the *BR-TC Traces Builder* is applied for the *BR-Feat Traces Builder*, but the Oracle used by the *Recovery Traceability Evaluator*. We also used the same effectiveness metrics as in the first study: $Precision$, $Recall$, and F_2-$Score$, the *REI* coefficient, and the *Goodness* scale.

A.4 Results and Discussion

A.4.1 General Evaluation

In this section, we report and discuss the results for the context of traceability between bug reports and features, which are summarized by the bar chart in Figure A.2.

RQ-1A: Which technique presents the best effectiveness in the context of traceability between bug reports and system features?

We have observed that, in the context of Mozilla Firefox artifacts, the LSI technique is the most effective, with the highest F_2-$Score$ (49.1%) in average, despite the existing tendency of replacement by the BM25 technique in existing search engines, such as the Apache Lucene[1]. LSI presented the second best $Recall$ (76.3%), and its $Precision$ (28.3%) is the best one nearly ten percent ahead of BM25's (17.5%), which is the second best. Even with a higher $Recall$, the BM25 did not achieved a better performance than the LSI in terms of F_2-$Score$, which evaluates $Precision$ and $Recall$ jointly and with an appropriate balancing between the two metrics for traceability recovery purposes, giving more importance to

[1]https://lucene.apache.org

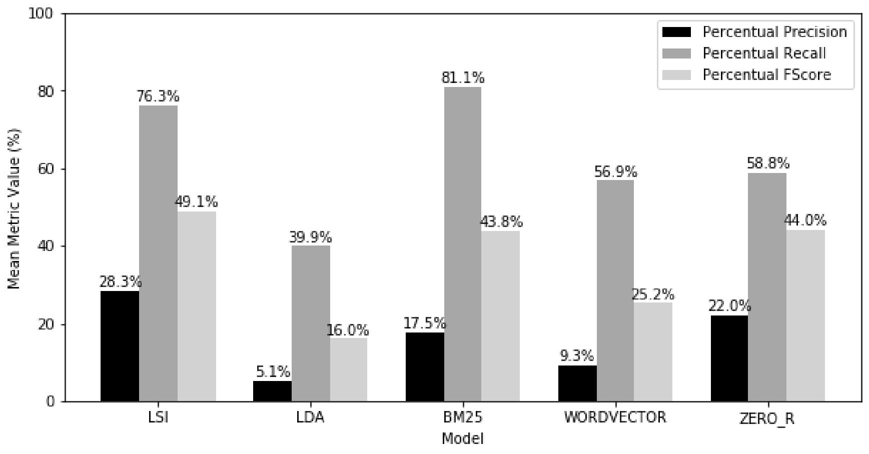


Figure A.2: BR-Feat Traceability Recovery Results

higher $Recall$ scores than to higher $Precision$ scores.

LDA presented the lowest F_2-$Score$ (16.0%). Confirming previous studies conclusions, the overall effectiveness of LDA is lower than LSI's [34; 25], in terms of $Recall$ and $Precision$. In the final result, LDA presented $Recall$ of 39.9% and $Precision$ of 5.1%. Analyzing the created topics, we checked that the LDA technique was not able to represent most of the system features as we wished, and it presented an odd tendency to classify almost every bug report as related to three system features: *New Awesome Bar*, *Windows 10 Compatibility*, and *WebGL Compatibility* (similarity score above 0.8). The description field of these system features influenced this technique behavior: common words from the bug reports content appear with a considerable frequency into these features descriptions, such as "Firefox," "page," "address," "bar," "web," and "URL.", so the technique attributed similar topics to them, what generated a high similarity score. LSI and BM25 techniques were able to better deal with such common words, which leveraged their results in comparison to LDA's.

On the other hand, although the Word Vector technique is promising for Natural Language Processing (and IR in general), its effectiveness is limited to the context of the training *corpora* from where it extracts the tokens. The training data used for creating the vector space model representation in our study is very different from general software engineering textual data: the corpora is created based on texts collected from comments, blogs and news on the web. This mismatch may negatively impact its prediction power and, in turn, its effectiveness in the traceability activity. Still, despite this drawback, the Word Vector model presented reasonable effectiveness in terms of $Recall$ (56.9%), but not in terms of $Precision$ (9.3%) and F_2-$Score$ (25.2%), if compared to the others techniques. Also, we have noticed that the mean similarity value considering every pair (feature, bug report) in the Word Vector similarity matrix is 0.86 and standard deviation 0.058, which means that the technique attributes high values of similarity between the majority of pairs (feature, bug report), that behavior and its consequences are more explored in following sections.

When we look at the effectiveness achieved by the baseline classifier (ZeroR), we see a significative performance, earned only predicting the *majority class* as explained in Section 5.4.4. By indicating only positive links for the relationships between a bug report and the *New Awesome Bar* system feature (major class), the ZeroR classifier had a $Precision$ of 22.0%.

The ZeroR $Precision$ turns the obtained $Precision$ of three out of four studied techniques unacceptably low – only the LSI technique had a $Precision$ above the baseline, which defines a minimum reference value to evaluate the techniques. Although we did not discard the other techniques because of that, once the $Precision$ is not the most important metric in our context and other metrics such as $Recall$ and F_2-$Score$ have more relevance in our analysis.

Continuing our explanation, we observed an identical phenomenon for the F_2-$Score$ metric, only the LSI reached a greater F_2-$Score$ if compared with the baseline. The BM25 F_2-$Score$ may be considered identical to the baseline one. However, when we look at the $Recall$ scores, the LSI and BM25 techniques achieved bigger $Recall$ scores than the baseline predictor, and were able to return the vast majority of relevant links between the bug reports and system features.

Surprisingly, the ZeroR baseline classifier achieved an *Acceptable* level of *Goodness*. The ZeroR F_2-$Score$ was 44%, therefore higher than the minimum value of 42.85% to be considered *Acceptable*. Analyzing the *Goodness* scale through the F_2-$Score$s obtained, we can see the LSI (49.1%) and BM25 (43.8%) also achieved an *Acceptable* level of *Goodness*, considering the mean of the combinations of Top Values and Similarity Values, as demonstrated in the bar plots of Figure A.2.

Deepening our analysis, we plotted PR-Curves with the $Precision$ and $Recall$ values

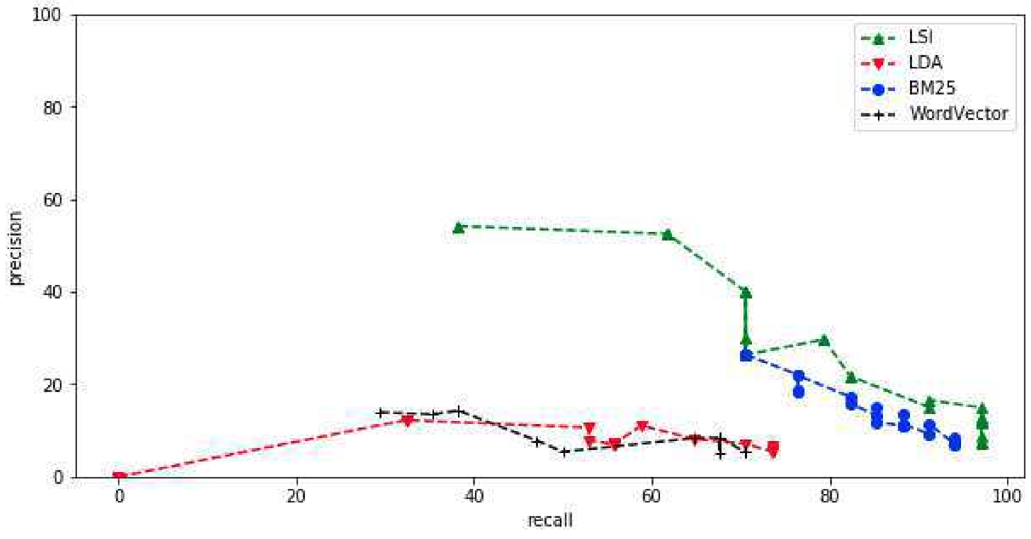
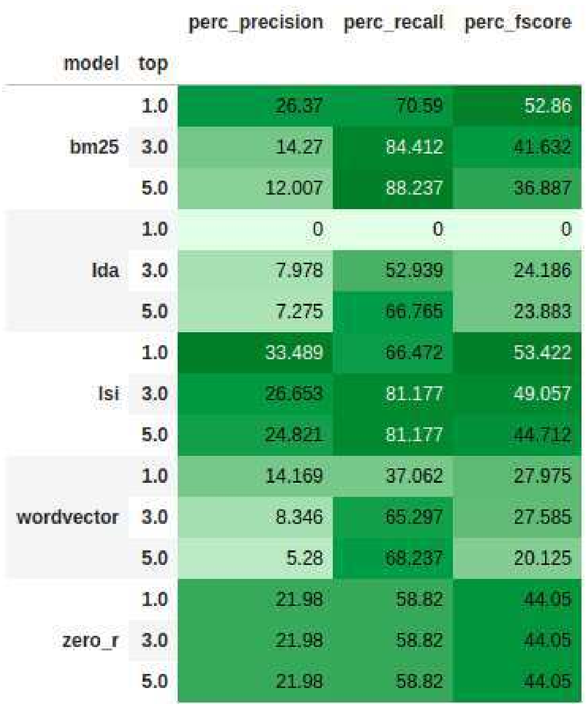


Figure A.3: PR-Curves of each technique – BR-Feat Context

obtained by each technique which are shown in Figure A.3. A PR-Curve in our context gives an instantaneous view of the effectiveness of all IR and DL techniques when compared with each other: the technique with a larger area under the curve is the best one [6]. In our study, the LSI technique is the best in this scenario and the state-of-the-art BM25 technique is the second best. We can observe that LSI obtains higher $Precision$ and $Recall$ values than BM25 in most of the cases. While LDA and Word Vector presented the lowest effectiveness, with the Word Vector slightly better than LDA.

When we aggregate the metrics by Top Value (1,3,5), as is shown in Figure A.4, we are able to compare the effect of the Top Value over the techniques performance – the darker green cells represent higher scores. We see only the LSI technique maintains a $Precision$ above the baseline (ZeroR classifier) independently of the Top Value and metric considered ($Precision$ percentage – **perc_precision**, $Recall$ percentage – **perc_recall**, and F_2-$Score$ percentage – **perc_fscore**). Note the BM25 technique only obtained scores ($Precision$ and $Recall$) above the baseline for the Top 1, and LDA and Word Vector techniques just in some cases and not for both metrics.

Next, we extend our analysis and evaluate the results for two different scenarios, so we better understand the effects of different Top Values and Similarity Thresholds over the technique effectiveness: (i) in the first scenario (Scenario I), we selected the case where the Similarity Threshold is 0.0, so the number of returned documents is maximized and the $Recall$



model	top	perc_precision	perc_recall	perc_fscore
bm25	1.0	26.37	70.59	52.86
	3.0	14.27	84.412	41.632
	5.0	12.007	88.237	36.887
lda	1.0	0	0	0
	3.0	7.978	52.939	24.186
	5.0	7.275	66.765	23.883
lsi	1.0	33.489	66.472	53.422
	3.0	26.653	81.177	49.057
	5.0	24.821	81.177	44.712
wordvector	1.0	14.169	37.062	27.975
	3.0	8.346	65.297	27.585
	5.0	5.28	68.237	20.125
zero_r	1.0	21.98	58.82	44.05
	3.0	21.98	58.82	44.05
	5.0	21.98	58.82	44.05

Figure A.4: Performance of techniques aggregated by Top Value

is favored over the $Precision$. (ii) in the second scenario (Scenario II), we evaluated the techniques using a Similarity Threshold of 0.9, such value tends to leverage the $Precision$ over $Recall$ metric.

A.4.2 Scenario I – Similarity Threshold 0.0

We start the analysis of this scenario by evaluating the traces that were captured by all techniques – characterizing the easiest ones – and, in the other hand, the no captured traces (not captured by any technique) – characterizing the hardest ones. Table A.1 summarizes these results.

Top	Not Captured Traces	Traces Captured by All
1	$\frac{7}{34} = 20.59\%$	$\frac{0}{34} = 0.00\%$
3	$\frac{0}{34} = 0.00\%$	$\frac{18}{34} = 52.94\%$
5	$\frac{0}{34} = 0.00\%$	$\frac{23}{34} = 67.65\%$

Table A.1: Captured and Not Captured Traces – All Techniques – Scenario I – Study I

Traces Missed by All Techniques Considering Top Value 1 we have that seven out 34 traces were not captured by any of the techniques, this represents 20.58% of the oracle's traces. The number of missed traces are mainly due to reasons: (i) the bug reports are related with more than one feature, so the Top 1 cut forces the exclusion of many of them, and (ii) the keywords characterizing the bug reports did not lead to a sufficient similarity score to bond them to the correct Mozilla's system features.

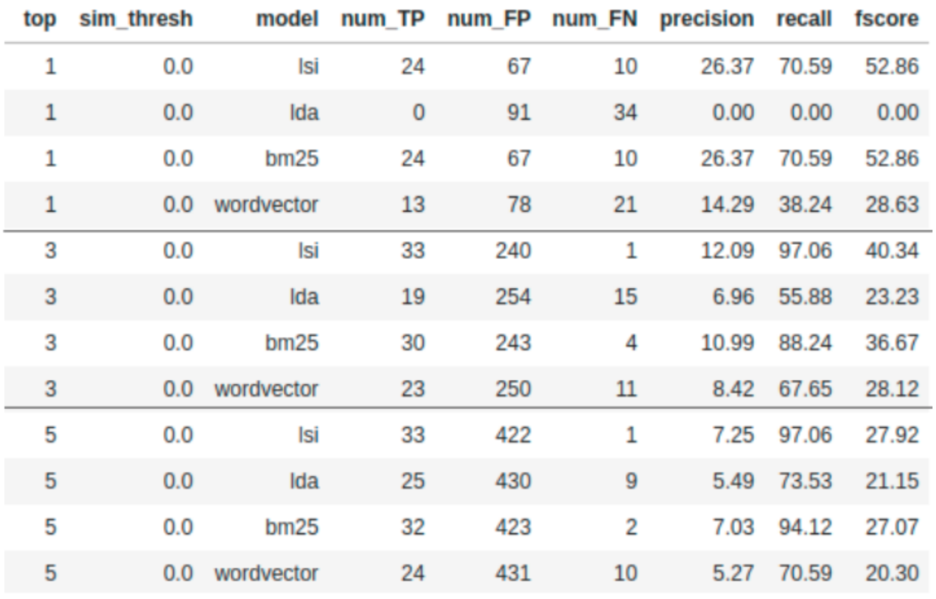
The first case occurs with the bug report 1357458 (*"After Customization - typed text in the Awesome bar doesn't correspond with the text from One-Off-Searches bar"*), which is linked with the system features *New Awesome Bar* and *Browser Customization*, but the Top 1 forces the recovery of only one of the traces – the one with the first feature.

Whereas the second case happened with the other six bug reports. In these cases, we detected that some bug reports have a description that misguided the techniques due to the presence of some keywords, such as "windows", that are more used to describe features, such as *Windows 10 Compatibility*, comparing with the similarity score attributed to the correct features. For example, the LSI technique attributed a similarity between the bug report 1318903 (*"[Windows 7 and below] Fullscreen window controls not shown with dark themes, close button has broken "red square" hover state"*) and *Windows 10 Compatibility* of 0.6470, while it attributed 0.4889 to the *Browser Customization* which is the correct system feature to be linked with this bug report.

Traces Captured by All Techniques Analyzing in the other extreme, we see that, for Top 1, zero percent of the traces were captured by all techniques. This demonstrates a relative difference between the techniques which belong to different families [5] – each one represented here by one technique – and that have distinct similarity scoring functions and core algorithms. When we observe the same scenario for Top 3 and 5, we have that 52.94% and 67.65%, respectively, of the traces are captured by all the techniques, these are considerable results and suggest the set of techniques is capable of recover the entire set of true traces, although such hypothesis needs further studies.

Techniques Evaluation

In this section, we evaluate each technique and explore some of the obtained results of traceability recovery extending the analysis considering each one of the Top Values (1,3,5) and the Similarity Threshold (0.0) in relation to the True Positives (TP), False Positives (FP) and False Negatives (FN) that characterize each technique. Figure A.5 details the obtained results.

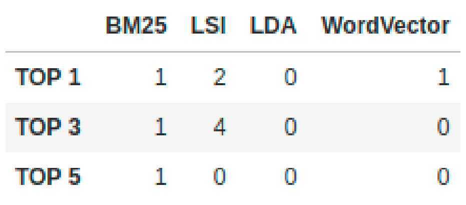


top	sim_thresh	model	num_TP	num_FP	num_FN	precision	recall	fscore
1	0.0	lsi	24	67	10	26.37	70.59	52.86
1	0.0	lda	0	91	34	0.00	0.00	0.00
1	0.0	bm25	24	67	10	26.37	70.59	52.86
1	0.0	wordvector	13	78	21	14.29	38.24	28.63
3	0.0	lsi	33	240	1	12.09	97.06	40.34
3	0.0	lda	19	254	15	6.96	55.88	23.23
3	0.0	bm25	30	243	4	10.99	88.24	36.67
3	0.0	wordvector	23	250	11	8.42	67.65	28.12
5	0.0	lsi	33	422	1	7.25	97.06	27.92
5	0.0	lda	25	430	9	5.49	73.53	21.15
5	0.0	bm25	32	423	2	7.03	94.12	27.07
5	0.0	wordvector	24	431	10	5.27	70.59	20.30

Figure A.5: Traceability Recovery Results for Scenario I – Extra Study

True Positives (TP) The technique with more exclusive identified true positives – traces correctly identified by only it – is LSI and the second best is BM25, although the difference is not expressive it is enough to grant some advantage to the first technique. Figure A.6 details the number of true positives exclusively identified by each technique for all top values considered. The LSI technique was able to recover the most of true positive traces with a $Recall$ of 97.06% at Top 3 and 5 – missed only 1 out of 34 traces. Also, the BM25 obtained significant results, very close to the LSI's, achieving 94.12% of $Recall$ at Top 5. However, the LSI's $Precision$ is equal or greater than BM25's in all Top Values.

In terms of *Goodness* analyzed using the F_2-$Score$ reference values (see Section 5.4.3 at Chapter 5), we see the LSI and BM25 techniques only obtained an *Acceptable* level of *Goodness* in Top 1, where the F_2-$Score$ of both is 52.86% in average.



	BM25	LSI	LDA	WordVector
TOP 1	1	2	0	1
TOP 3	1	4	0	0
TOP 5	1	0	0	0

Figure A.6: Number of true positives exclusively identified

The LDA technique results were surprisingly poor at Top 1, where it did not recover any true trace, so that its $Precision$, $Recall$, and F_2-$Score$ were zero for this case. The main reason for that is the preference of the technique for the system feature *WebGL Compatibility*. The technique attributed the highest value of similarity that it has calculated between all system features to this feature specifically for almost every bug report. However, how this system feature was not linked with any of the selected bug reports in the oracle, then the technique was not able to recover none relevant traces. We were not able to explain this particular behavior of the technique, although we have identified that the three preferred system features are all strongly related to a single topic into the technique's internal structure.

Considering the Word Vector technique, we noticed a reasonable performance, although not sufficient to classify its level of *Goodness* as *Acceptable* in any Top Value, the technique only recover a maximum of 70.59% of relevant traces at Top 5 – its best performance. The technique presented a similar behavior to the identified in the Second Empirical Study (Chapter 5), where it attributed high similarity scores to nearly all pair of a bug report and system feature, in this case. This is problematic once demonstrates the technique was not able to distinguish the textual differences between the documents being analyzed. We recommend – as we did in the previous study – to use a weighting scheme into the techniques algorithms, so these differences may be better detected, such as happens in techniques how LSI and BM25.

False Positives (FP) Confirming the poor effectiveness of the LDA technique in terms of $Precision$, this technique had the highest number of false positives between all techniques. How was described in Section A.4.1, the LDA technique had the odd behaviour of relating almost every bug report to the features *New Awesome Bar*, *Windows 10 Compatibility*, and *WebGL Compatibility*. This behavior was responsible for the high number of false positives, as we can verify in the Venn diagrams in Figure A.7.

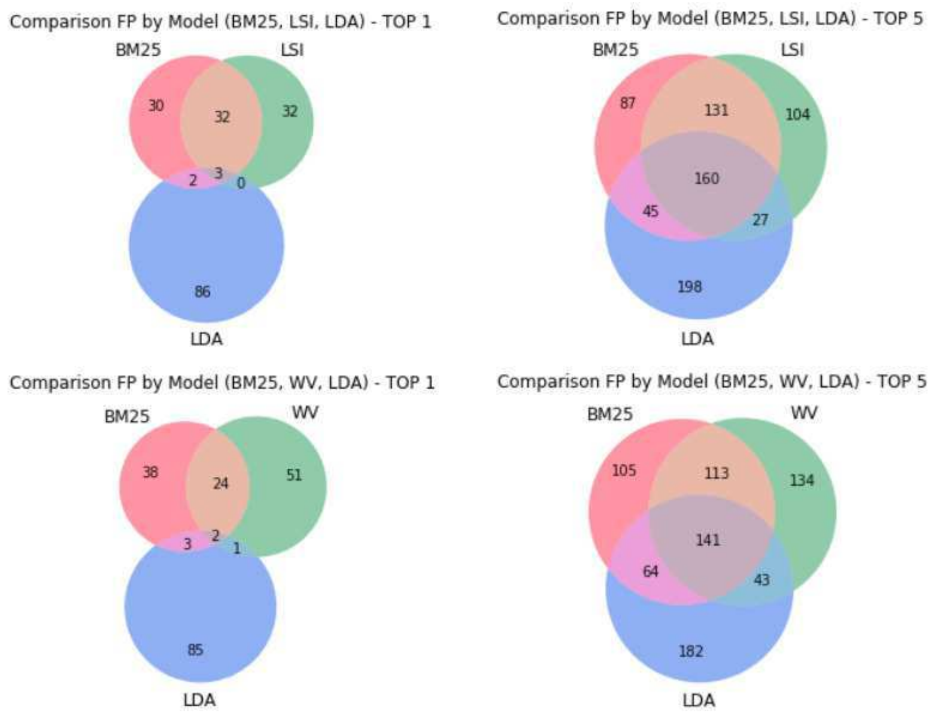


Figure A.7: Comparison of False Positives

Whereas the BM25 and LSI techniques have a large number of common false positives traces – 32 for Top 1 and 131 for Top 5 – what indicates some proximity between the technique's behavior when considering only a fixed cut (Top Value) if compared with the other two techniques (LDA and Word Vector). Additionally, the diagrams reveal a considerable number of false positives that are common for all techniques, even larger than the intersection between LSI's and BM25's false positives, which suggests all of them are making the same mistakes and probably a variable cut (Similarity Threshold) may be beneficial for the $Precision$'s improvement of each technique.

The presence of a variable Similarity Threshold may be a requirement for using the Word Vector technique successfully, considering the problem described in Section A.4.1 – the technique tends to attribute high values of similarity to almost every pair (bug report, system feature) –, once the variable cut can help the technique to better distinguish the pair of documents. However, the adequate values of cut should be calculated separately for each technique, so it can be adapted to the scale of similarity values of each technique. Some studies proposed and experienced with identical hypothesis [2; 9;

10] identifying the correctness of it, corroborating our discoveries into the context of traceability studied by us.

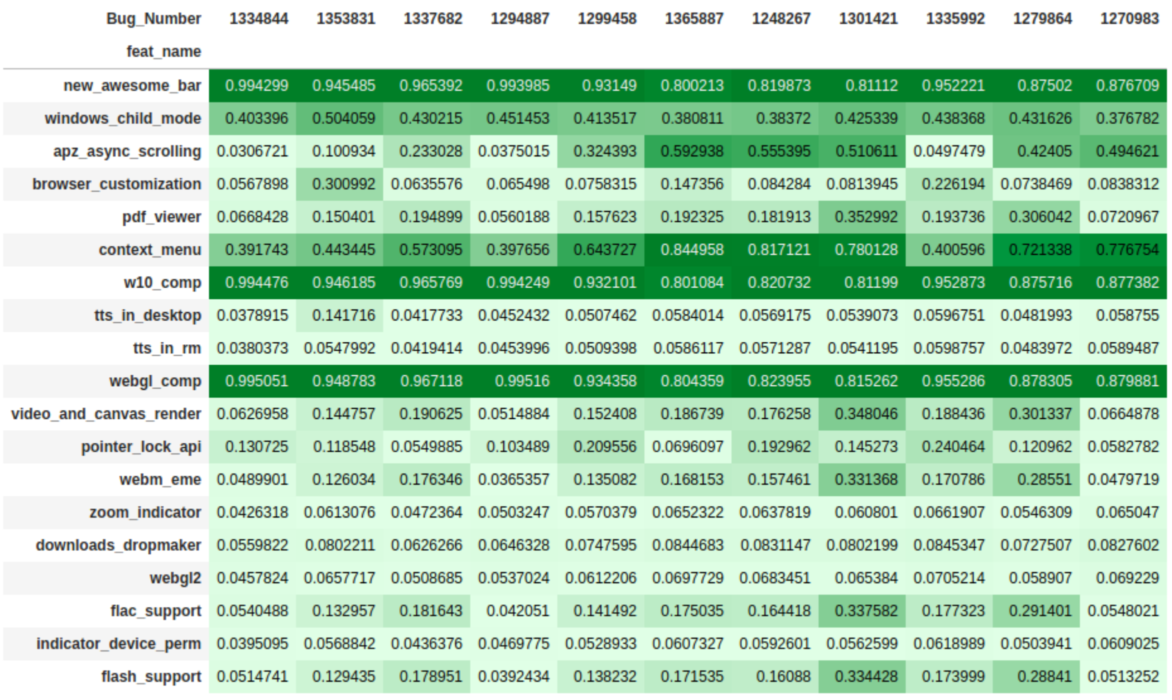
False Negatives (FN) When we look at the false negatives, in average, the poorest results were again produced by the LDA technique, which had the highest number of false negatives (34 out of 34) in Top 1 (0.0% $Recall$), and second lowest (9 out of 34) for Top 5 (73.53% $Recall$). In the first case, the technique presented 11 exclusive false negatives. Eight out of these 11 false negatives were related to the system feature *New Awesome Bar* and the remaining three to the *Context Menu* feature.

Despite the tendency of the technique to relate almost every bug report to three system features and between these is the *New Awesome Bar*, we checked that in these eight cases, the negative traces were due to lack similarity of the respective eight bug reports, which were not able to reach enough similarity scores with *New Awesome Bar* feature when compared with the other two technique's preferred system features (*WebGL Compatibility* – ***webgl_comp*** and *Windows 10 Compatibility* – ***w10_comp***). Figure A.8 shows a subset of LDA's similarity matrix with these eight bug reports and highlighted similarity values: darker the cell, higher the similarity value. Observe the clear tendency of the technique relate all bug reports with the three preferred system features (***w10_comp***, ***webgl_comp***, and *textbfnew_awesome_bar*), but in prejudice of the *New Awesome Bar* feature when we make a Top 1 cut.

The LSI, BM25 and Word Vector techniques had no exclusive false negatives for Top 1. However, Word Vector (WV) presented a single exclusive false negative in Top 5. Checking the Venn diagrams in Figure A.9, we observed the Word Vector (WV) and LDA have most of the false negatives in common.

A.4.3 Scenario II – Similarity Threshold 0.9

Alike in the first scenario in this section, we evaluate each technique obtained traceability results in relation to True Positives (TP), False Positives (FP) and False Negatives (FN). The considered Top Values are the same as in the first scenario (1,3,5), but the fixed Similarity Threshold is 0.9. We expected the $Precision$ scores improve and the $Recall$ may decrease with this set up.



Bug_Number feat_name	1334844	1353831	1337682	1294887	1299458	1365887	1248267	1301421	1335992	1279864	1270983
new_awesome_bar	0.994299	0.945485	0.965392	0.993985	0.93149	0.800213	0.819873	0.81112	0.952221	0.87502	0.876709
windows_child_mode	0.403396	0.504059	0.430215	0.451453	0.413517	0.380811	0.38372	0.425339	0.438368	0.431626	0.376782
apz_async_scrolling	0.0306721	0.100934	0.233028	0.0375015	0.324393	0.592938	0.555395	0.510611	0.0497479	0.42405	0.494621
browser_customization	0.0567898	0.300992	0.0635576	0.065498	0.0758315	0.147356	0.084284	0.0813945	0.226194	0.0738469	0.0838312
pdf_viewer	0.0668428	0.150401	0.194899	0.0560188	0.157623	0.192325	0.181913	0.352992	0.193736	0.306042	0.0720967
context_menu	0.391743	0.443445	0.573095	0.397656	0.643727	0.844958	0.817121	0.780128	0.400596	0.721338	0.776754
w10_comp	0.994476	0.946185	0.965769	0.994249	0.932101	0.801084	0.820732	0.81199	0.952873	0.875716	0.877382
tts_in_desktop	0.0378915	0.141716	0.0417733	0.0452432	0.0507462	0.0584014	0.0569175	0.0539073	0.0596751	0.0481993	0.058755
tts_in_rm	0.0380373	0.0547992	0.0419414	0.0453996	0.0509398	0.0586117	0.0571287	0.0541195	0.0598757	0.0483972	0.0589487
webgl_comp	0.995051	0.948783	0.967118	0.99516	0.934358	0.804359	0.823955	0.815262	0.955286	0.878305	0.879881
video_and_canvas_render	0.0626958	0.144757	0.190625	0.0514884	0.152408	0.186739	0.176258	0.348046	0.188436	0.301337	0.0664878
pointer_lock_api	0.130725	0.118548	0.0549885	0.103489	0.209556	0.0696097	0.192962	0.145273	0.240464	0.120962	0.0582782
webm_eme	0.0489901	0.126034	0.176346	0.0365357	0.135082	0.168153	0.157461	0.331368	0.170786	0.28551	0.0479719
zoom_indicator	0.0426318	0.0613076	0.0472364	0.0503247	0.0570379	0.0652322	0.0637819	0.060801	0.0661907	0.0546309	0.065047
downloads_dropmaker	0.0559822	0.0802211	0.0626266	0.0646328	0.0747595	0.0844683	0.0831147	0.0802199	0.0845347	0.0727507	0.0827602
webgl2	0.0457824	0.0657717	0.0508685	0.0537024	0.0612206	0.0697729	0.0683451	0.065384	0.0705214	0.058907	0.069229
flac_support	0.0540488	0.132957	0.181643	0.042051	0.141492	0.175035	0.164418	0.337582	0.177323	0.291401	0.0548021
indicator_device_perm	0.0395095	0.0568842	0.0436376	0.0469775	0.0528933	0.0607327	0.0592601	0.0562599	0.0618989	0.0503941	0.0609025
flash_support	0.0514741	0.129435	0.178951	0.0392434	0.138232	0.171535	0.16088	0.334428	0.173999	0.28841	0.0513252

Figure A.8: Highlighted LDA's Similarity Matrix

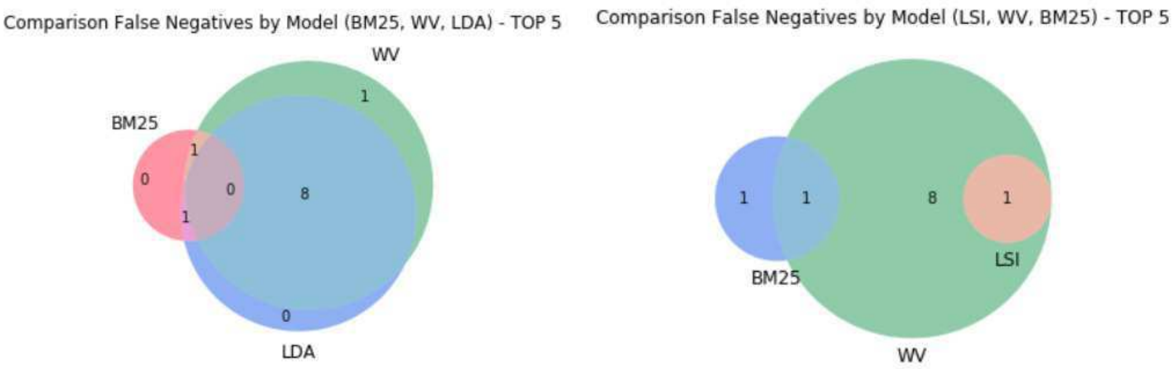


Figure A.9: False Negatives Comparison

Top	Not Captured Traces	Traces Captured by All
1	$\frac{10}{34} = 29.41\%$	$\frac{0}{34} = 0.00\%$
3	$\frac{6}{34} = 17.65\%$	$\frac{5}{34} = 14.71\%$
5	$\frac{5}{34} = 14.71\%$	$\frac{5}{34} = 14.71\%$

Table A.2: Captured and Not Captured Traces – All Techniques – Scenario II – Study I

Traces Missed by All Techniques Once in this second scenario we had a more restrictive similarity threshold than in the first scenario, the number of traces which were not captured by any of the techniques is larger than in the first scenario. One of the reasons is that in the first we had only a fixed cut (Top Value) as limiting factor for the number of returned documents (features) for each query (bug report), but for this second scenario, besides it, we had a high similarity threshold too.

Table A.2 details the number and proportion of captured and not captured traces. In Top 1 nearly 30% of the traces were not recovered by any technique. Analyzing the 10 not captured traces, we see the following distribution: 2 traces should be linked with the *Browser Customization* feature, another 2 with the *Downloads Dropmaker*, four with the *New Awesome Bar*, one with the *PDF Viewer*, and one with the *Zoom Indicator*. Whereas we have the distribution of traces/number of bug reports by feature as stated by Table A.3.

Feature	**Num_BRs**
New Awesome Bar	20
Downloads Dropmaker	4
Indicator for Dev. Permissions	3
Context Menu	3
Browser Customization	2
PDF Viewer	1
Zoom Indicator	1

Table A.3: Distribution of Bug Reports by Feature

In this second scenario, we have the problems identified in the first scenario with practically the same bug reports as source artifacts and the new problem that is the high similarity threshold of 0.9. Bug reports linked with more than one system feature were not linked to all their system features due to the Top 1 cut, which selects only the feature with higher similarity. An example of that is the bug report 1305195 (*"In private browsing mode, zoom level indicator is unreadable when dark developer edition theme is in use"*), which is linked with the *Browser Customization* and *Zoom Indicator* features, but is not linked with both at Top 1, and the techniques, such as Word Vector, that attributes high levels of similarity between documents are not able to recover this link until Top 5, characterizing its low $Precision$ scores.

The main difference between the results obtained at Top 1 to the ones obtained at Top 3 and 5 is the set of traces associated with the *New Awesome Bar*, which were all captured by the BM25 or the Word Vector techniques at Top 3 and 5. The first may be favored by the normalization applied to the similarity scores – which leveraged the similarity score relative to the *New Awesome Bar* feature to values above 0.9 –, while the second was favored by its high similarity scores, which in this case returned the correct features.

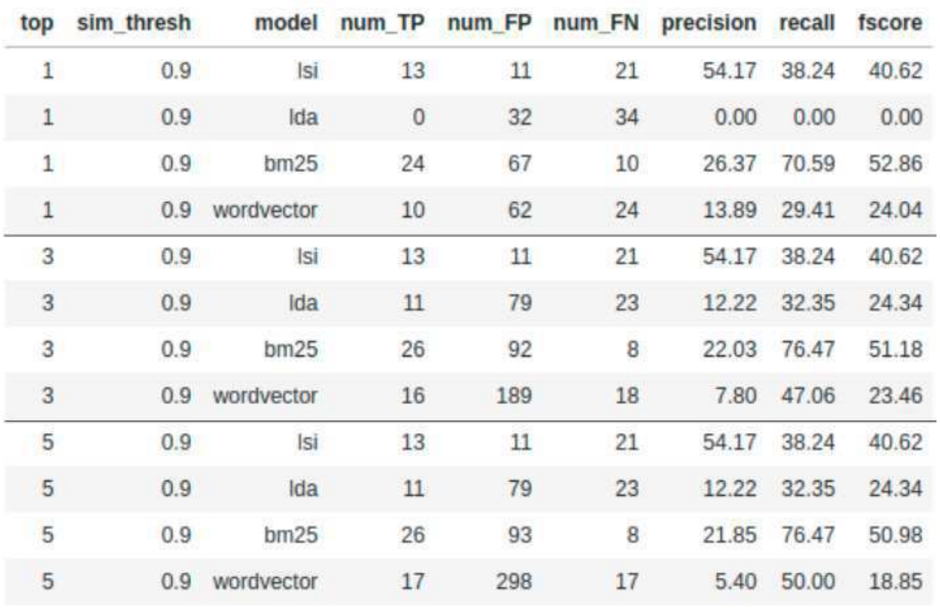
Other problems identified were (i) the trace relative to *PDF Viewer* was not recovered, and (ii) both traces associated with *Downloads Dropmaker* were not recovered. We estimate the reason for the first case was the *PDF Viewer* short description, which hampers the tracking by techniques. On the other hand, the reason for the second was the short bug description. At this point, becomes clear that in a traceability recovery process between bug reports and features would greatly benefit from a common vocabulary and a guide for writing both artifacts.

Traces Captured by All Techniques We also observed the lower number of traces captured by all techniques simultaneously when comparing with the first scenario. For Top 1, 0.0% of the traces were recovered by all techniques at the same time, while for Top 3 and 5, the same five traces were recovered, which represents 14.71%. The lower number is expected once the high similarity threshold (0.9) imposes bigger restrictions to relate a given bug report with some system feature. Added to this, we have that each technique belongs to a different family into the set of IR and DL techniques, so they use different similarity functions and then attribute high similarity scores in distinct manners, which decreases the intersection between the sets of recovered traces.

Techniques Evaluation

In this section, we analyze the effectiveness of the different techniques considering each of the Top Values studied separately. Figure A.10 shows the number of true positives (num_TP), number of false positives (num_FP), number of false negatives (num_FN), the $Precision$, the $Recall$, and the F_2-$Score$ obtained.

In this second scenario was expected the $Precision$ scores would improve in general, while the $Recall$ scores would diminish in comparison to the first studied scenario. Except for the Word Vector, this phenomenon was observed in all techniques. However, we noticed



top	sim_thresh	model	num_TP	num_FP	num_FN	precision	recall	fscore
1	0.9	lsi	13	11	21	54.17	38.24	40.62
1	0.9	lda	0	32	34	0.00	0.00	0.00
1	0.9	bm25	24	67	10	26.37	70.59	52.86
1	0.9	wordvector	10	62	24	13.89	29.41	24.04
3	0.9	lsi	13	11	21	54.17	38.24	40.62
3	0.9	lda	11	79	23	12.22	32.35	24.34
3	0.9	bm25	26	92	8	22.03	76.47	51.18
3	0.9	wordvector	16	189	18	7.80	47.06	23.46
5	0.9	lsi	13	11	21	54.17	38.24	40.62
5	0.9	lda	11	79	23	12.22	32.35	24.34
5	0.9	bm25	26	93	8	21.85	76.47	50.98
5	0.9	wordvector	17	298	17	5.40	50.00	18.85

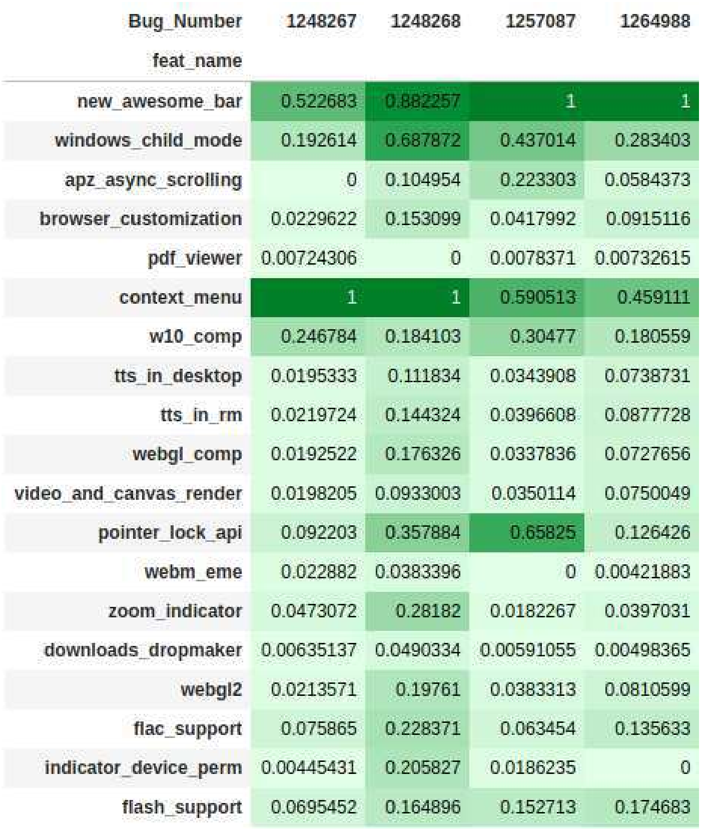
Figure A.10: Traceability Recovery Results for Scenario II – Extra Study

two similar values of $Precision$ and $Recall$ when comparing the two scenarios for two techniques: LDA had $Precision$ and $Recall$ of 0.00% in both scenarios for Top 1; and BM25 had $Precision$ of 26.37% and $Recall$ of 70.59% in both scenarios for Top 1.

The LDA's behavior is explained by the same reason identified in the first scenario – the technique has a preference for tracing almost every bug report for three system features, which are not correct ones. The technique's similarity values are mostly smaller than 0.9, even for the preferred three system features. This lead to the repetition of the poor results of the first scenario, especially for Top 1.

While the BM25's identical results are explained by the normalization of its similarity scores, where for Top 1 just the feature with a similarity score normalized to the value "1" is returned and which happens to be the same independent of the similarity threshold applied. Figure A.11 shows a subset of the BM25 highlighted similarity matrix. Observe the presence of the darker green cells with a 1 indicating the trace with the highest similarity score after normalization.

Whereas the Word Vector technique, as was already explained, tends to attribute high values of similarity scores for the evaluated pairs (bug report, system feature), but even the highest value of similarity threshold used in our study (0.9) was not enough for increasing the technique's $Precision$, which on the contrary diminished in Top 1 and 3 in comparison



Bug_Number	1248267	1248268	1257087	1264988
feat_name				
new_awesome_bar	0.522683	0.882257	1	1
windows_child_mode	0.192614	0.687872	0.437014	0.283403
apz_async_scrolling	0	0.104954	0.223303	0.0584373
browser_customization	0.0229622	0.153099	0.0417992	0.0915116
pdf_viewer	0.00724306	0	0.0078371	0.00732615
context_menu	1	1	0.590513	0.459111
w10_comp	0.246784	0.184103	0.30477	0.180559
tts_in_desktop	0.0195333	0.111834	0.0343908	0.0738731
tts_in_rm	0.0219724	0.144324	0.0396608	0.0877728
webgl_comp	0.0192522	0.176326	0.0337836	0.0727656
video_and_canvas_render	0.0198205	0.0933003	0.0350114	0.0750049
pointer_lock_api	0.092203	0.357884	0.65825	0.126426
webm_eme	0.022882	0.0383396	0	0.00421883
zoom_indicator	0.0473072	0.28182	0.0182267	0.0397031
downloads_dropmaker	0.00635137	0.0490334	0.00591055	0.00498365
webgl2	0.0213571	0.19761	0.0383313	0.0810599
flac_support	0.075865	0.228371	0.063454	0.135633
indicator_device_perm	0.00445431	0.205827	0.0186235	0
flash_support	0.0695452	0.164896	0.152713	0.174683

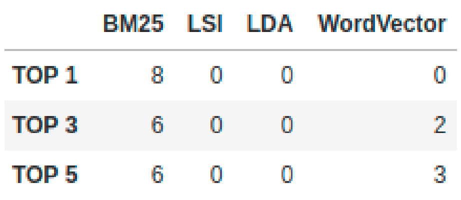
Figure A.11: BM25 Similarity Matrix Subset

to the first scenario. A variable similarity threshold adapted for the technique's similarity score scale could be applied for improving the results, as explained in Section A.4.2, this solution may be the most appropriate for this technique so its effectiveness improves.

True Positives (TP) Comparatively to the other techniques, the BM25 retrieved the largest number of exclusive true positives, as shows the Figure A.12. Gaining from the other techniques by a large margin, considering that are only 34 true traces, it is able to recover exclusively eight traces (23.52%) that the other techniques are not able for Top 1. However, if the similarity threshold applied would be 0.8 instead of 0.9, the LSI technique would have hit the seven out of these eight traces.

Whereas for Top 3 and 5, all BM25's exclusive traces would be also recovered by the LSI. The proximity between the techniques can be verified here, but the application of *normalization* of the BM25's similarity scores again played an essential rule for this favorable result over the LSI. In terms of $Recall$, the BM25 technique had the highest scores for all the

Top values studied (all cases above 70%). On the other hand, we cannot claim the same for the LSI, whose $Recall$ scores did not surpass 40%. Again this results from the normalization of the BM25's similarity scores.



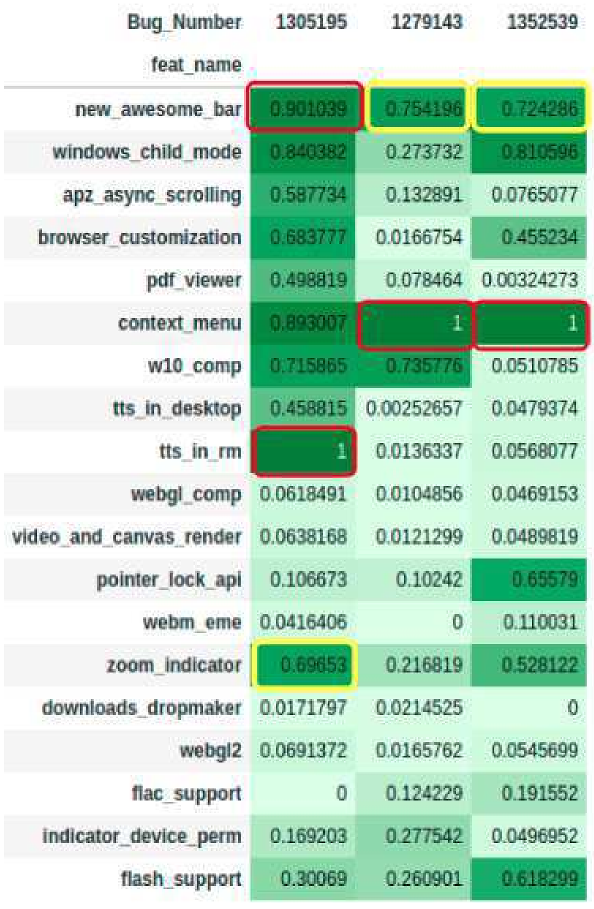
	BM25	LSI	LDA	WordVector
TOP 1	8	0	0	0
TOP 3	6	0	0	2
TOP 5	6	0	0	3

Figure A.12: Number of true positives exclusively identified

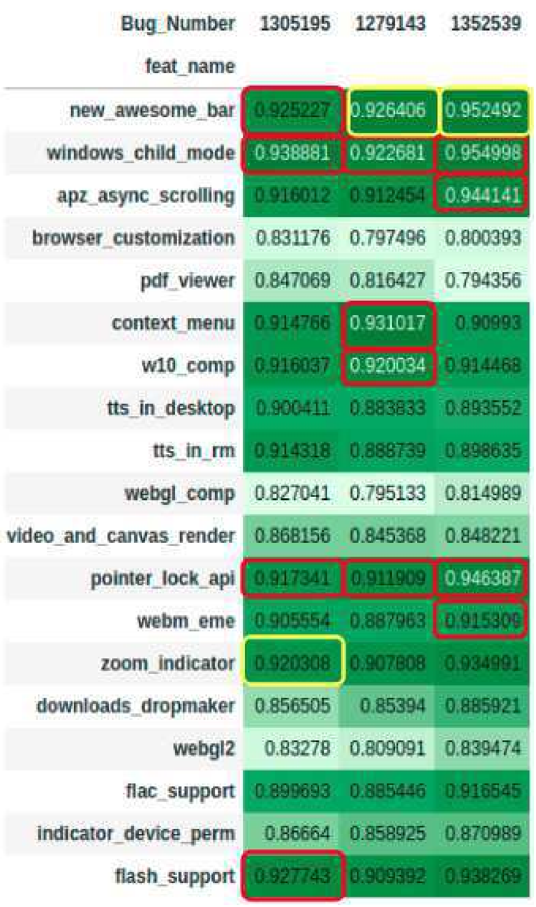
The Word Vector technique had the second best $Recall$ scores and the second largest number of true positives for Top 3 and 5, cases in which it had, respectively, two and three exclusive true positive traces recovered. The technique was able to maintain a reasonable level of $Recall$ (47.06% and 50.0%, respectively) due to the high values of similarity presented in its similarity matrix, but the same reason is responsible for the lower values of $Precision$, as explained at the beginning of this section.

Comparing BM25 and Word Vector effectiveness, the best ones in terms of the number of TP, we are able to highlight some points. We noticed the presence of ambiguous words in the bug reports that may have misguided the BM25 technique. One example of that is the bug report 1305195 (*"In private browsing mode, zoom level indicator is unreadable when dark developer edition theme is in use"*), which refers to "private browsing mode" and should be linked with the features *Browser Customization* and *Zoom Indicator*. However, the reporter used the word "mode" three times, which made the technique to relate this bug report with the feature *Text to Speech in Reader Mode*. The ambiguity of the words is one of the main problems faced by natural language processing techniques in general, and this includes Information Retrieval techniques as BM25. This problem is addressed by Deep Learning techniques, such as Word Vector, where the context is considered in the Vector Space Model creation, so the technique is able to face ambiguity issues with some success.

Figure A.13 shows the highlighted similarity matrices of BM25 and Word Vector techniques with the Top 5 returned traces highlighted. In the left we see the BM25 similarity matrix, the true traces are highlighted in yellow, while the remaining traces are in red. In the right with have the correspondent Word Vector similarity matrix with the same bug re-



Bug_Number	1305195	1279143	1352539
feat_name			
new_awesome_bar	0.901039	0.754196	0.724286
windows_child_mode	0.840382	0.273732	0.810596
apz_async_scrolling	0.587734	0.132891	0.0765077
browser_customization	0.683777	0.0166754	0.455234
pdf_viewer	0.498819	0.078464	0.00324273
context_menu	0.893007	1	1
w10_comp	0.715865	0.735776	0.0510785
tts_in_desktop	0.458815	0.00252657	0.0479374
tts_in_rm	1	0.0136337	0.0568077
webgl_comp	0.0618491	0.0104856	0.0469153
video_and_canvas_render	0.0638168	0.0121299	0.0489819
pointer_lock_api	0.106673	0.10242	0.65579
webm_eme	0.0416406	0	0.110031
zoom_indicator	0.69653	0.216819	0.528122
downloads_dropmaker	0.0171797	0.0214525	0
webgl2	0.0691372	0.0165762	0.0545699
flac_support	0	0.124229	0.191552
indicator_device_perm	0.169203	0.277542	0.0496952
flash_support	0.30069	0.260901	0.618299



Bug_Number	1305195	1279143	1352539
feat_name			
new_awesome_bar	0.925227	0.926406	0.952492
windows_child_mode	0.938881	0.922681	0.954998
apz_async_scrolling	0.916012	0.912454	0.944141
browser_customization	0.831176	0.797496	0.800393
pdf_viewer	0.847069	0.816427	0.794356
context_menu	0.914766	0.931017	0.90993
w10_comp	0.916037	0.920034	0.914468
tts_in_desktop	0.900411	0.883833	0.893552
tts_in_rm	0.914318	0.888739	0.898635
webgl_comp	0.827041	0.795133	0.814989
video_and_canvas_render	0.868156	0.845368	0.848221
pointer_lock_api	0.917341	0.911909	0.946387
webm_eme	0.905554	0.887963	0.915309
zoom_indicator	0.920308	0.907808	0.934991
downloads_dropmaker	0.856505	0.85394	0.885921
webgl2	0.83278	0.809091	0.839474
flac_support	0.899693	0.885446	0.916545
indicator_device_perm	0.86664	0.858925	0.870989
flash_support	0.927743	0.909392	0.938269

Figure A.13: BM25 (left) vs Word Vector (right) Similarity Matrices – Top 5 in Red and True Traces in Yellow – Similarity Threshold 0.9

ports. Observe the smaller number of returned traces by the BM25 technique due to the high similarity threshold of 0.9.

False Positives (FP) The Venn diagrams in Figure A.14 provides details about the number of false positives traces associated with each technique. Through the analysis of the diagrams, we see the techniques with the largest numbers of false positives are BM25 and Word Vector in both Top 1 and 5. However, if the increasing trajectory of the BM25 number of false positives is much smaller than Word Vector's, whose number of false positives is nearly multiplied by a factor of five when comparing Top 1 (62) and Top 5 (298).

Another interesting fact about the false positives is that in all studied cases of Top Values all the LSI's false positives are included into BM25's false positives, and we visualize the considerable difference between the two most precise techniques in the Venn diagrams,

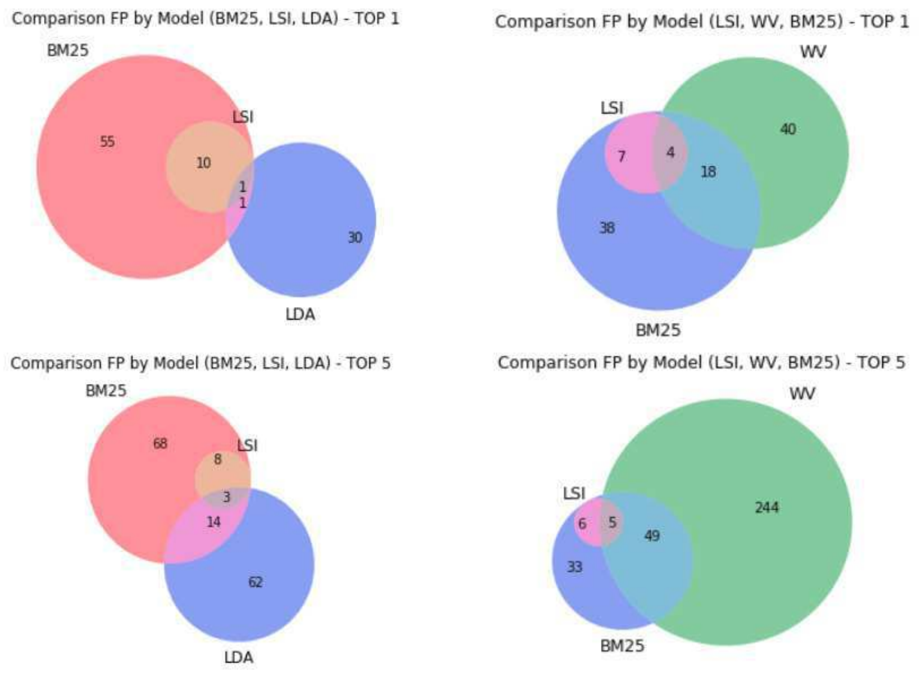


Figure A.14: Comparison of false positives

where the LSI set appears into the BM25 ones.

Top	BM25	LSI	LDA	Word Vector
1	37	0	30	40
3	35	0	58	139
5	28	0	51	233

Table A.4: Number of exclusive false positives

The number of exclusive false positives is detailed in Table A.4. Observe how the number of false positives from the Word Vector technique becomes much bigger in comparison to the other ones with the increase in the Top Value. It is worth noting also how the LDA technique prefers the *WebGL Compatibility* feature, corresponding to 28 exclusive positive traces from LDA in Top 1 and 5. We were not able to identify the reason for this technique behavior.

False Negatives (FN) A general view of the false negatives associated with each technique is shown by Figure A.15. The Venn diagrams allow us to clearly see the poor effectiveness of the LDA technique if compared with the others. Note how it is the bigger set of false negative traces in all cases comparatively the other techniques.

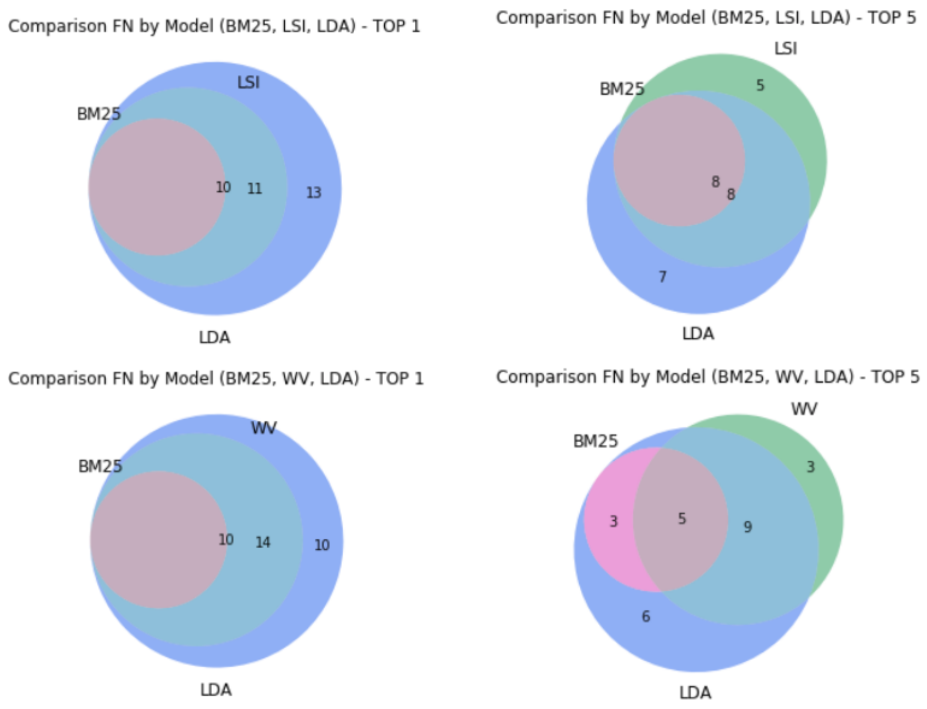


Figure A.15: Comparison of false negatives

In the context of the exclusive false negatives, we have the results summarized in Table A.5. The LDA technique is the one with the largest number of exclusive false negatives between all techniques. In this case, the preference of the technique by three system features, how was already explained, combined with the similarity threshold of 0.9 were responsible for the technique's poor effectiveness. We can observe that six out of seven (85.7%) of exclusive false negative traces for Top 1 were associated with the system feature *New Awesome Bar*, and the technique would be able to correctly return them if the similarity threshold would be 0.8 or the fixed cut (Top Value) would be bigger than 1. This is confirmed when we look at the cases of Top 3 and 5 which have a smaller number of exclusive false negatives.

Top	**BM25**	**LSI**	**LDA**	**Word Vector**
1	0	0	7	0
3	0	3	4	1
5	0	3	4	1

Table A.5: Number of exclusive false negatives

Whereas the BM25 technique, how we see in Figure A.15, had no exclusive false negative traces for any of the Top values – all BM25's false negatives were also erroneously indicated by other techniques. While the Word Vector had only one exclusive false negative trace for Top Values 3 and 5, which was relative to *New Awesome Bar*, but the technique traced with no system feature due the similarity threshold value of 0.9 – the calculated similarity value between the bug report *1335992* and the *New Awesome Bar* feature was 0.8910, which did not allow the linking between the bug report and the system feature.

In the same way, the LSI technique was not able to correctly trace the three exclusive false negatives indicated in Table A.5. The calculated similarity values for the respective three bug reports and the correct system feature (*New Awesome Bar*) were around 0.7, *i.e* below the threshold of 0.9, not recovering the traces.

A.4.4 Best Similarity Threshold Value

In this section we evaluate the hypothesis of existence of a best similarity threshold value into the range of values studied, identically we made in Chapter 5 (Section 5.5.5). Figures A.16 and A.17 show the obtained results of $Precision$, $Recall$, and F_2-$Score$ considering the variation of the similarity threshold and for each Top Value (1,3,5). The Figure A.16 depicts the effects of similarity threshold variation over the LSI and LDA techniques, while the Figure A.17 shows the effects over the BM25 and Word Vector techniques.

We can visualize in each plot the $Precision$ (in blue), the $Recall$ (in green), the F_2-$Score$ (in brown), and the three reference values for F_2-$Score$ (in red), so we can determine the *Goodness* level of each technique. The bottom line is relative to the *Acceptable* level, the middle line to the *Good* level, and the top line to the *Excellent* level of *Goodness*.

Through the analysis of Figures A.16 and A.17, we see two of the techniques presented satisfactory levels of *Goodness*: the LSI and the BM25. The first one obtained a *Good* level for similarity thresholds 0.6, 0.7, and 0.8 in all Top Values – observe the F_2-$Score$ passing the reference value relative to *Good* (55.26%) with the best similarity threshold being 0.7. While the BM25 achieved an *Acceptable* level of *Goodness* for all similarity thresholds at Top 1, and maintained this level for similarity thresholds 0.6 to 0.9 in Top 3 and 0.7 to 0.9 in Top 5.

Both techniques could be used for traceability recovery between bug reports and system features in a semi-automatized process, where an analyst or engineer is involved in the

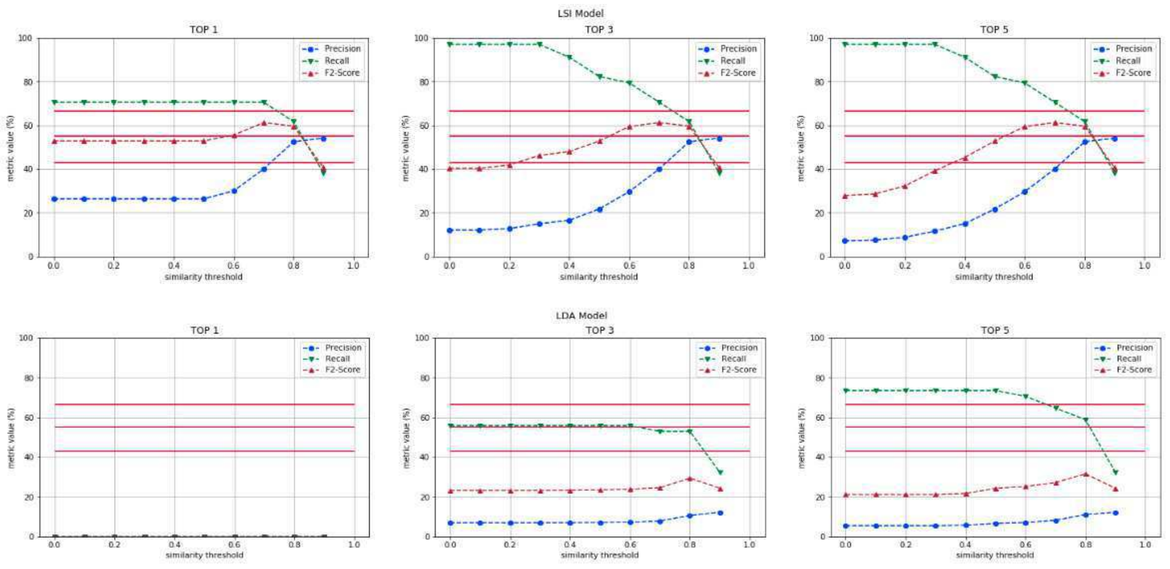


Figure A.16: LSI and LDA Similarity Threshold Variation – Extra Study

process of recovering and checking the traces recovered by an appropriate software tool. Although, as we have seen, the LSI presented better results and is recommended for these kinds of artifacts used.

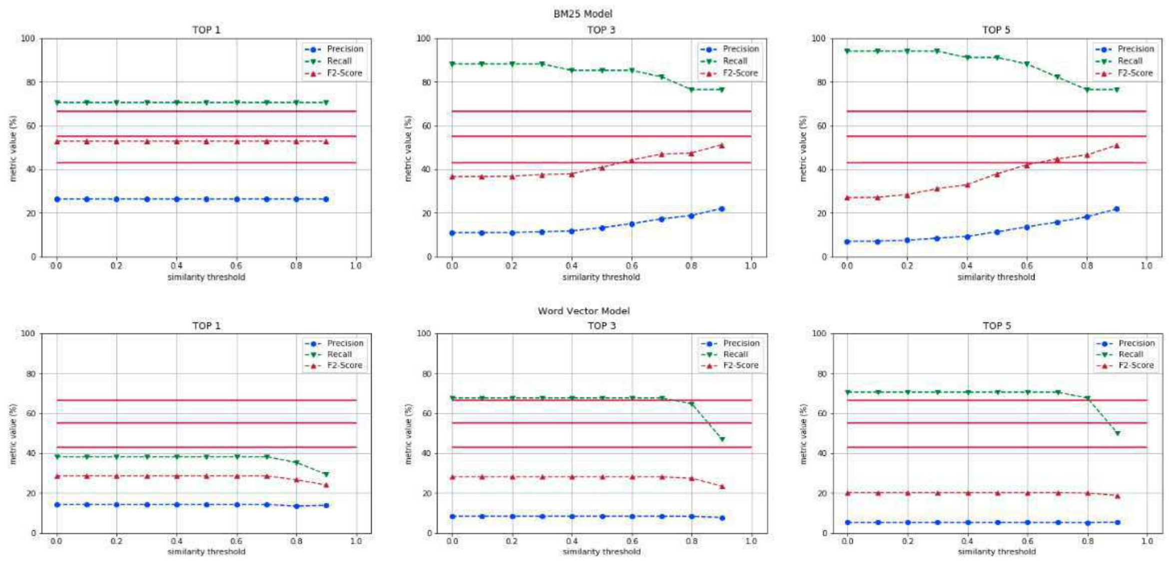


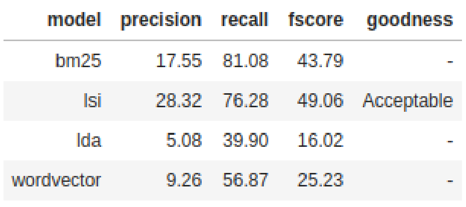
Figure A.17: BM25 and Word Vector Similarity Threshold Variation – Extra Study

Continuing the analysis of the Figures A.16 and A.17, we checked the LDA had very poor performance for every combination of similarity threshold and Top Value, especially at Top 1, where it obtained zero in every considered metric. In the other Top Values (3 and

5), the technique obtained reasonable $Recall$ scores, although the $Precision$ was very low. The same can be said about the Word Vector in these Top Values, the techniques have a small difference in their general effectiveness, but in Top 1 the Word Vector achieved greater effectiveness than LDA's.

A.4.5 Goodness Scale

Hayes *et al.* [20] developed the referred *Goodness Scale* from their work with different types of software artifacts to evaluate the general effectiveness of multiple traceability techniques and estimate the level of work required from the analyst to analyze the returned traces. The scale is based on $Precision$ and $Recall$ values, and we adopted the metric as defined by the authors, just extending the reference metrics to include the F_2-$Score$ (see Section 5.4.3 in Chapter 5).



model	precision	recall	fscore	goodness
bm25	17.55	81.08	43.79	-
lsi	28.32	76.28	49.06	Acceptable
lda	5.08	39.90	16.02	-
wordvector	9.26	56.87	25.23	-

Figure A.18: Goodness scale by each technique – Extra Study 1

Figure A.18 shows the obtained results in average $Precision$, $Recall$, and F_2-$Score$ values for each technique. We checked the LSI again stands out over the other techniques with an Acceptable *Goodness* level. Despite the high $Recall$ results of the BM25 technique, it did not achieve a minimum $Precision$ (20%) to be considered Acceptable, when we look only at the average values. Whereas the other techniques did not present satisfactory results.

These results allow us to conclude that only the LSI technique is feasible for use in the context of a real project, as the Mozilla Firefox, and for the task of traceability recovery between bug reports and features when considering the average results of the many Top Values and Similarity Thresholds. However, as was demonstrated in the previous section A.4.4, the BM25 also is an *Acceptable* technique for some combinations of Top Value and Similarity Thresholds. So, ultimately, both LSI and BM25 are eligible for use in traceability recovery tasks aiding human analysts in tracing links between bug reports and system features.

A.4.6 Recovery Effort Index – *REI*

In this section, we report and analyze the *REI* values obtained for each technique considering all Top Values (1,3,5) and all Similarity Thresholds ([0.0, 0.1, ..., 0.9]). The $Precision$ value of the oracle produced with the answers exclusively from the volunteers (see Chapter 3 and Section 5.4.2), in relation to the oracle produced only with the expert's answers, is 36.53%.

Model	**REI**
BM25	1.57
LSI	0.67
LDA	1.99
Word Vector	4.69

Table A.6: *REI* values

Table A.6 summarizes the obtained *REI* values. Once the basis of REI calculation stands over the $Precision$ values of each technique and LSI had the highest average $Precision$ value between all techniques, we see the consequences of this reflected on the *REI* values. The LSI has the lowest recovery effort index, which means the analyst or engineer saves more time analyzing the traceability results – including the time of discard false positives – using this technique than any other in the set of studied techniques. The closest technique to LSI is BM25, but yet the saved time is nearly double if compared with LSI's.

It is important to notice that we make a free association of *REI* values with the time required from the analysis, but this association still needs deeper study and we cannot attribute statistical significance to it without further study.

A.4.7 Lessons Learned

In this extra study – focusing on traceability between bug reports and system features – the techniques with the best effectiveness were LSI and BM25 in that order. These techniques presented a degree of feasibility for industrial application in projects such as the Mozilla Firefox, adopting a reference scale of *Goodness*, focusing in the analysis of $Precision$, $Recall$, and F_2-$Score$; and using determined combinations of Top Values and Similarity Thresholds. A factor of success of these techniques is their weighting schemes and scoring functions, which were able to capture the important keywords presented in the bug reports and system features and correctly link them. However, we have indications that the effort required from the analyst in using the LSI technique is lower than the BM25's required effort.

In the other hand, the LDA and Word Vector presented the poorest effectiveness. The first – based on the creation of topics – was not able to correctly characterize the bug reports topics, so they could be related to the right system features. Whereas the second was not capable of capturing the nuances between the bug reports and system features, attributing high values of similarity for almost every pair of a bug report and system feature, probably to the lack of a weighting scheme into the technique. This behavior granted a considerable technique's $Recall$, but a low $Precision$.

We estimate the Word Vector would achieve better results if trained exclusively with textual data originated from the software engineering context as bug reports, test cases, and use cases. The final model would be able to capture semantic and syntactical relationships between the words that are particular for this specific context and the vector space built would be more representative of this specific context. A current limitation for doing this is the lack of available data, such training requires a massive amount of data.