Automated Software Engineering (2021) 28:8
https://doi.org/10.1007/s10515-021-00287-w

Traceability recovery between bug reports and test cases-a Mozilla Firefox case study

Guilherme Gadelha[1] · **Franklin Ramalho**[1] · **Tiago Massoni**[1]

Received: 12 October 2020 / Accepted: 24 June 2021 / Published online: 7 July 2021

Abstract

Automatic recovery of traceability between software artifacts may promote early detection of issues and better calculate change impact. Information Retrieval (IR) techniques have been proposed for the task, but they differ considerably in input parameters and results. It is difficult to assess results when those techniques are applied in isolation, usually in small or medium-sized software projects. Recently, multilayered approaches to machine learning, in special Deep Learning (DL), have achieved success in text classification through their capacity to model complex relationships among data. In this article, we apply several IR and DL techniques for investing automatic traceability between bug reports and manual test cases, using historical data from the Mozilla Firefox's Quality Assurance (QA) team. In this case study, we assess the following IR techniques: LSI, LDA, and BM25, in addition to a DL architecture called Convolutional Neural Networks (CNNs), through the use of Word Embeddings. In this context of traceability, we observe poor performances from three out of the four studied techniques. Only the LSI technique presented acceptable results, standing out even over the state-of-the-art BM25 technique. The obtained results suggest that the semi-automatic application of the LSI technique – with an appropriate combination of thresholds – may be feasible for real-world software projects.

Keywords Bug reports · System features · Test cases · Traceability · Information retrieval · Deep learning

✉ Guilherme Gadelha
guilherme@copin.ufcg.edu.br

Franklin Ramalho
franklin@computacao.ufcg.edu.br

Tiago Massoni
massoni@computacao.ufcg.edu.br

[1] Computing and Systems Department (DSC), Federal University of Campina Grande (UFCG), Campina Grande, Paraiba, Brazil

1 Introduction

Software development and testing involve, as sub-products, textual artifacts, such as bug reports, test cases, requirements documents, besides the source code itself. The produced artifacts are interrelated, and tracking those relationships may bring benefits to software teams. This is especially important for *requirements*, to which several artifacts at different 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" (Gotel and Finkelstein 1994).

An effective traceability recovery process has significant repercussions over software development activities (Guo et al. 2017). In such a scenario, bugs reported by developers, testers, final users, or stakeholders could be automatically selected and prioritized for bug fixing and testing tasks. Also, development teams could precisely estimate the impact of stakeholders' changes if information about the affected artifacts is available for decision-makers, reducing the involved risks for the project. The same information could be used for budget prediction once more data are available to estimate the teams' sizes and the number of hours required for bug fixing. Another benefit is the reduction of the learning curve required from new team members. Usually, the projects do not have up-to-date documentation about the decisions made during the process, so 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. Traceability recovery tools may reduce this learning process and speed up integrating these new members for more critical activities into the project.

Scalable traceability requires automation; if manually maintained, it becomes an error-prone and expensive task (Hayes et al. 2007; Dekhtyar et al. 2007). Traceability tools and techniques emerged in response to that demand, allowing traceability links between any textual artifacts to be quickly recovered and analyzed. Nevertheless, 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 (Antoniol et al. 2002), using the Vector Space Model (VSM) technique. Ensuing, many other studies were developed using other techniques, such as Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), and Best Match 25 (BM25). Borg et al. (2014) verified in a systematic literature review the most common techniques are LSI and VSM, although the BM25 is the state-of-the-art technology in the field.

IR techniques differ considerably in terms of input parameters and results. Solutions proposed by previous research (Hayes et al. 2007; Canfora and Cerulo 2006; Oliveto et al. 2010; Lormans and Van Deursen 2006) for traceability recovering between software artifacts explore a specific technique, 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 sound basis for requirements traceability, which is even more complex as the most cited studies focus on small

and medium-sized software projects (see discussion regarding related work in Sect. 8). Furthermore, studies using Machine Learning (ML) and Deep Learning (DL) techniques/models for requirements traceability have been carried out, focusing on either requirements identification (Dekhtyar and Fong 2017), number of remaining traceability links estimation (Falessi et al. 2017), or traceability links prediction (Guo et al. 2017). However, they should be compared with IR techniques for assessing their effectiveness.

Previous studies address traceability recovery between many types of software artifacts. Still, we have noticed only a few studies *tracing bug reports to manual test cases* (Borg et al. 2014), even though manual test cases often are the most up-to-date documentation of the system and the only available source of system requirements (Bjarnason et al. 2016), especially in agile development teams that do not shift their focus to automated tests (Sabev and Grigorova 2015), being naturally referred to by bug reports. In fact, a few studies have shown that software repositories often lack automatic test cases (Minelli and Lanza 2013). One previous related study addresses traceability between requirements and bug reports (Yadla et al. 2005); Hemmati *et al.* (Hemmati and Sharifi 2018) investigate IR techniques for predicting manual test case failure; Merten *et al.* (Merten et al. 2016) analyzed variations of five IR techniques for traceability recovery between bug reports. Still, only one study deals with traceability between manual test cases and bug reports (Kaushik et al. 2011). See Sect. 8 for more details on these two last studies.

In summary, to the best of our knowledge, there are no clear indications about the most effective 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 approaches relating bug reports and test cases have limitations on the variety of studied techniques and the evaluation's depth.

To fill that knowledge gap, in this article we ran a case study that applies a set of IR and DL techniques to recover traceability links between bug reports and manual test cases (often known as *test scripts*), using publicly-available historical data from the Mozilla Firefox's development team[1]. Despite the comprehensiveness of this dataset, bug reports are not explicitly linked to manual test cases; for establishing a ground truth to evaluate the recovery techniques, we used system features as intermediate artifacts. These system features allowed us to group test cases, helping us generate a ground truth to evaluate each technique. We applied the following techniques: Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), Best Match 25 (BM25), and two different versions of the Word Vector technique. 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 artifacts used in genuine and open source projects such as the Mozilla Firefox.

[1] https://www.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.

Fig. 1 Example of Firefox bug report

We have observed that LSI presents the most competitive results compared with other IR or DL techniques for traceability recovery between bug reports and test cases. Comparative analysis suggests LSI is superior to a baseline classifier (VSM), achieving an acceptable level of *Goodness*, considering the obtained scores of *Precision*, *Recall*, and F_2-*Score*, for some specific combinations of Top Values and Similarity Thresholds.

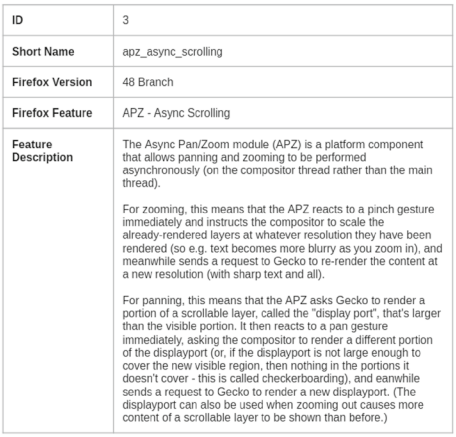
This article is organized as follows: Sect. 2 defines important concepts to better understand our work and the applied techniques; Sect. 3 describes the approach developed for linking the analyzed artifacts; whereas Sect. 4 explains the oracle building process; and Sect. 5 exposes the case study and achieved results. The validity threats to our conclusions are explained in Sect. 7, while the related work is discussed in Sect. 8. Conclusions are summarized in Sect. 9.

2 Background

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

2.1 Bug reports

A Bug Report describes any system failure, either identified by a user or automatically reported by the system (in the case of crashing bugs) (Fazzini et al. 2018). A bug report offers details about a failure identified in order to help developers investigate and fix the bug reported if its presence is confirmed (Lee 2016). Bugs occur due to either implementation faults or specification nonconformances that are 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



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.)

Fig. 2 Example of system feature from Mozilla Firefox

behavior, among other data (Davies and Roper 2014). Figure 1 shows an example of a bug report from the Mozilla Firefox repository[2].

A properly-reported bug includes a clear and detailed problem description. Also, 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 (Zimmermann et al. 2010).

In Fig. 1, we identify attributes which qualify a bug report and contribute to its 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 (Kun Chen et al. 2005). System features improve communication efficacy offering a common vocabulary, which demands less cognitive effort for understanding, in comparison to individual requirements (Passos et al. 2013). The definition of system features creates a common ground so that every stakeholder can quickly understand the system operations.

A feature is commonly described by its *Name* and *Description*, but other 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 can be observed in Fig. 2[3] that shows the APZ—*Async Pan/Zoom* – system feature from Mozilla Firefox. This feature is related to the bug

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

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

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.

Fig. 3 Example of a manual test case for Mozilla Firefox

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 following fields were extracted to characterize a system feature: *ID*, *Short Name*, *Firefox Version*, *Firefox Feature* (Feature Name), and *Feature Description*.

2.3 Manual test cases

A Manual Test Case is a sequence of steps defined according to the stakeholder's requirements. The sequence of steps are executed manually into the produced software and the specified results are checked with the produced software outputs, so the tester can sign the test state as failed or successful (Sommerville 2010). Figure 3 shows an example of test case from the Mozilla Firefox, which is related with the previously presented Bug Report and System Feature. 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.

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 they match with the program outputs, then the test passes; otherwise, it fails. The *TC Number* is a unique ID for the test case. Especially in 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*. In this case, the manual test scripts are essential for software evolution and maintenance, allowing the detection of bugs before software is released to the final users. Also, scripts for manual test cases are easy to automate and facilitate the tests and bugs reproducibility. Besides that, manual scripts for test cases are the most up-to-date documentation of many systems in the industry, especially in agile contexts Bjarnason et al. (2016).

Fig. 4 Similarity matrix example

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

2.4 IR and DL techniques for traceability recovery

Information Retrieval techniques recover and rank a set of documents from a corpus, for a given query. Their output, in the context of traceability recovery, is a *similarity matrix* (Antoniol et al. 2002) between documents and queries. This matrix holds the similarity scores for each pair (document, query); scores are calculated according to the technique's core algorithm – assuming a similarity scale from zero to one (where one would denote two identical documents). In our work, bug reports are queries, while test cases are the documents. An example of a similarity matrix between bug reports and test cases is shown in Fig. 4. The green scale is correspondent to the level of similarity between the Bug Report row and the Test Case column; for example, the similarity score between Bug Report 1248268 and Test Case 5 is 0.441724.

To improve IR performance, techniques require one to preprocess both corpus and the set of queries, through the following stages: (i) tokenization of each document, removing blank spaces and punctuation; (ii) removal of stop words to discard articles, adverbs, and prepositions; (iii) application of *stemming* in each token, removing words suffixes (*informat*ion, *informat*ics, and *informat*ization would e treated as one token); and (iv) *lemmatization*, changing all verbs to the first person and present tense.

Deep Learning techniques also can be used for calculating similarity scores. Based on recent research (Dekhtyar and Fong 2017; Guo et al. 2017), we employ two versions of a Deep Learning technique as traceability recovery techniques (Sect. 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 (Borg et al. 2014) and literature (Blei et al. 2003; Oliveto et al. 2010; Robertson and Zaragoza 2009; Dekhtyar and Fong 2017; Guo et al. 2017).

One technique identified but not used in our study is the Jensen-Shannon model. We have not found either available source code with the model's implementation in our literature review nor existing implementation of it in available open-source libraries or frameworks. Additionally, a previous work demonstrated some equivalence between the LSI, VSM and Jensen-Shannon traceability results (Oliveto et al. 2010).

2.4.1 Latent Semantic Indexing

Latent Semantic Indexing (LSI) (Deerwester et al. 1990) is based on a vector space model (Borg et al. 2014; De Lucia et al. 2006; Dekhtyar et al. 2007). Each document in the corpus and each query are both vectorized. For this, it applies a specific weighting scheme, assigning the most relevant words of each document and query appropriate weights, in the searching and ranking process. LSI commonly applies the weighting scheme known as *tf-idf – term frequency-inverse document frequency*. The *tf-idf* formula is detailed in Eq. 1. Function $tf(t, d)$ yields the frequency of term t in document d, so the more the term appears in the document, the higher is *tf*. On the other hand, $idf(t, D)$ is the number of documents term t appears in the entire corpus D, so the rarer the term is, the higher its value.

$$tfIdf(t, d, D) = tf(t, d).idf(t, D) \tag{1}$$

The Eq. 2 details the smoothed *idf* formula, where N is the size of the corpus, and n_t is the number of documents in which term t appears. Since n_t value can be zero, the equation is corrected by summing 1 to the denominator.

$$idf(t, D) = \ln(\frac{N}{1 + n_t}) \tag{2}$$

Using the *term-by-document* matrix, whose content is *tf-idf*, as input, a mathematical dimensionality reduction method known as SVD (Singular Value Decomposition (Deerwester et al. 1990)) is applied, yielding new vectors representing documents and queries. This method optimizes LSI's effectiveness, speeding up search. The similarity score between each pair (document, query) is then determined by the cosine of the angle between the document and query vectors.

To illustrate LSI, we present an example with the bug report from Sect. 2.1. We relate it with three test cases – one from Sect. 2.3, and additional two that are displayed in Fig. 5.

We refer to the bug report and test cases used through their identifiers. The acronyms 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). The recovering process starts with preprocessing test cases and the bug report. LSI's application for traceability recovering is presented in Fig. 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.

Fig. 5 Test cases 13 and 60 used in our example

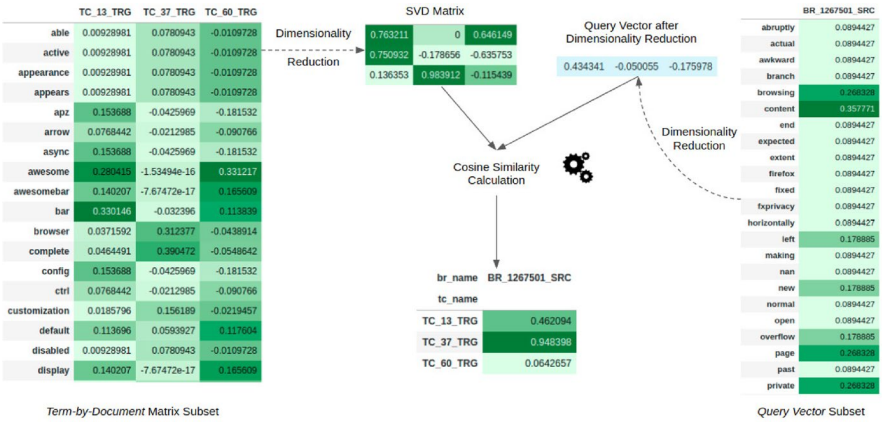


Fig. 6 LSI example

The *Term-by-Document Matrix* on the left-hand side of Fig. 6 is created with the terms presented in the test cases. The green scale indicates the most frequent terms in each document. Similarly, the query vector –based on the bug report —is depicted on the right-hand side. Weights are calculated using the *tf-idf* scheme; for this bug report, which contains 200 words, the word "apz" appears 5 times, thus receiving $tf = 5/200 = 0.025$. Now, assuming there is a corpus of 300 test cases, and "apz" appears in 35 of these, $idf = \ln(300/35) = 8.57$. Thus, *tf-idf* is the product of these quantities: $0.025 * 8.57 = 0.2142$ (Manning et al. 2009).

Next, SVD is applied over the matrices created, generating two matrices: *SVD 3x3 Matrix* from the *Term-by-Document* Matrix and another one from the *Query Vector* with dimensions 1x3. Then, cosine similarity is calculated between the line

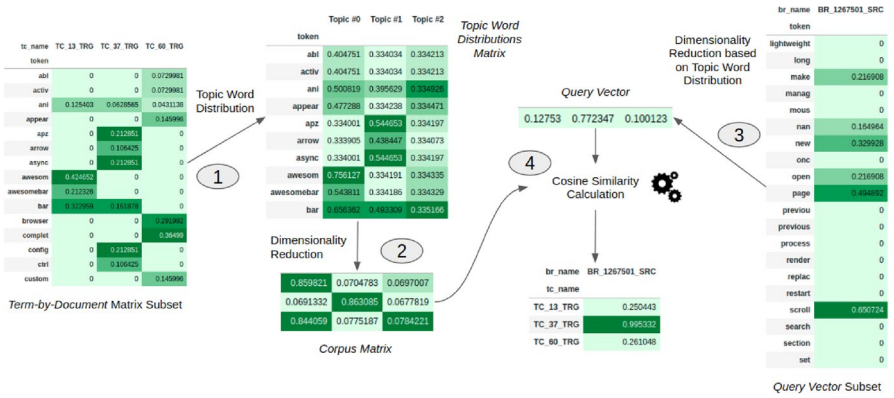


Fig. 7 LDA example

in the SVD matrix and the reduced query vector, resulting in the vector depicted at the center of Fig. 6. For this example, LSI was able to correctly recover Test Case 37 from the analysis of the bug report with a similarity score of 0.9483. Contrarily, low similarity scores (lighter green) assigned to the other two test cases, denoting a weak relationship.

2.4.2 Latent Dirichlet Allocation

The LDA technique (Blei et al. 2003) is a generative statistical model[4], in which each textual document is modeled as a set of *topics*. A Topic is characterized by a distribution over words, where each word has a distinct weight in that distribution allowing a human, through the analysis of the most relevant words, to attribute a semantic to each topic modeled by the technique. The estimated probabilities are relative to the following question: *What is the probability of a query* ***q*** *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, assigning it a specific distribution of topics. With the topics of a given query, the similarity scores between the query and the documents can be estimated. Several metrics can be used to calculate the similarity scores, such as the cosine of the angle between the vectors of each pair (document, query), as LSI. The difference is that the vectors here are vectors of probabilities (Dekhtyar et al. 2007).

An example of LDA, using the same test cases and bug report from the LSI example, is shown in Fig. 7. The left-hand side table shows tokens of each test case after *tf-idf* application; the result is a *Term-by-Document Matrix* (a subset is depicted in Fig. 7). Next, LDA's topic word distributions are created (Step 1). In this example,

[4] 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) (Y. Ng and Jordan 2002)

```
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
```

Fig. 8 LDA topics

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.

Through the analysis of the topics distributions, the technique successfully identifies the test cases associated with the system features: the first topic (*Topic #0)* is referring to the browser customization feature, whereas the second topic (*Topic #1)* indicates scrolling in the 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 Fig. 8, highlighting the core words.

After the topic word distribution calculation, a dimensionality reduction operation is applied, and a *Corpus Matrix* with dimensions 3x3 is generated (Step 2). A similar operation (Step 3) is applied over the bug report (query) vector, generating a reduced query vector with dimensions 3x1. Finally, the cosine similarity is determined for each line of the *Corpus Matrix* (test case) and the reduced query vector (bug report) (Step 4). The technique correctly recovers the related test case (37), with a high similarity score 0.9953, while assigning lower similarity scores to the other test cases. Figure 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 (Robertson and Zaragoza 2009). 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?* For that, scoring functions are used to rank the set of documents concerning each query.

The scoring function of the BM25 model can be generally described by Eqs. 3 and 4 (Canfora and Cerulo 2006). A document's score (d) concerning query q is calculated by Equation 3, in which t is each term in q and $W(t)$ is the weight of a specific term t for d. In Eq. 4, TF_t is the term frequency in 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. Variables k_1 and b are parameters, calibrating the effect of term frequency and of document length, respectively.

$$S_d(q) = \sum_{t \in q} W(t) \tag{3}$$

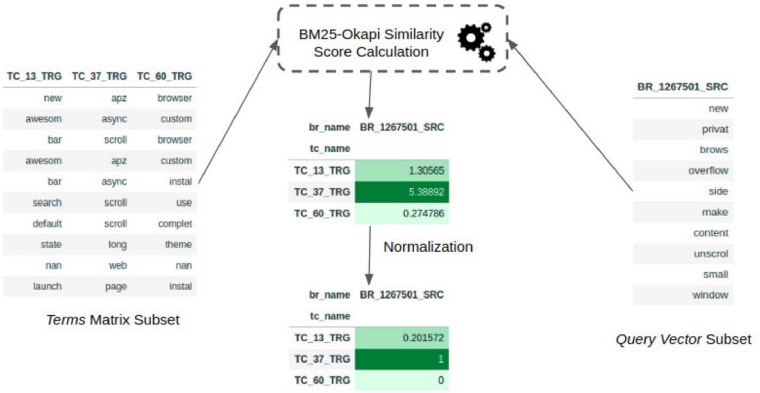


Fig. 9 BM25 example

$$W(t) = \frac{TF_t(k_1+1)}{k_1((1-b)+b.\frac{DL}{AVGDL})} log(\frac{N}{ND_t}) \tag{4}$$

We use the same bug report and test cases for illustrating BM25. Figure 9 shows the technique application. Each document in the corpus of test cases is preprocessed, having its tokens extracted into the *Terms Matrix Subset*, on the left-hand side. The same process is applied to the query (bug report), on the right-hand side. 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. To compare the BM25's similarity scores with other techniques, we apply the normalization of the scores for the scale [0,1], so the smallest score becomes 0, the higher becomes 1, and different values are calculated with these two reference values in the scale [0,1]. BM25 was able to recover the correct trace of the bug report with test case 37, while it assigns low values of similarity for the other test cases.

2.4.4 Word Vector

Deep Learning (DL) is a family of methods between the Machine Learning methods based on Artificial Neural Networks (Goodfellow et al. 2016). These networks characterize themselves for having a large number of hidden layers so that they are able of capturing many different patterns present in images, text corpora, and audio records data sets. Once the deep neural network is trained, it can recognize objects in images, translate texts between languages, and do speech recognition between many other applications.

Word Vector is a Deep Learning technique inspired by recently developed studies in the field (Dekhtyar and Fong 2017; Guo et al. 2017). A Word Embedding is a deep neural network trained based on large data sets of texts and which is capable of capture syntactic and semantic relations between the represented words. 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

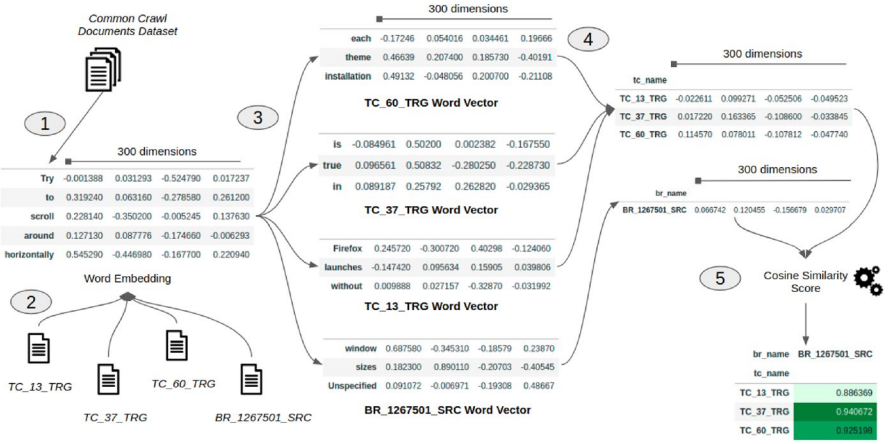


Fig. 10 Word vector example

and increasing computing capabilities to give support to these advancements. The release and dissemination of open-source libraries such as Google's word2vec[5] (Mikolov et al. 2013) facilitate their use as pre-trained models – trained on available data sets – or for training new word embeddings.

The word2vec library receives as input a large amount of text, such as the Common Crawl Dataset[6], which is a corpus of collected news, comments, and blogs on the web, and produces as output a vector space model, commonly with hundreds of dimensions. A vector represents each unique word (token) 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) or another type of Deep Neural Network. In the context of traceability, the trained neural network is available, possibly retrained with the source and target artifacts, so nuances from the domain of these textual documents can be captured 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 semantic inferences. For example, presenting the relationship *(Man,Woman)* for the model, and asking to the corresponding relationship for the word *King (King,?)*, the model is capable of correctly answering *(King,Queen)* (Mikolov et al. 2013).

For traceability recovery, word vectors can be used to measure the similarity between single words, but also between documents and queries like IR techniques. The example in Fig. 10, reusing the same bug report and test cases, is divided into five steps:

[5] https://github.com/svn2github/word2vec
[6] https://commoncrawl.org/

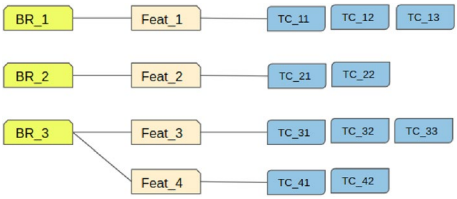


Fig. 11 Bug reports, system features and test cases relationships

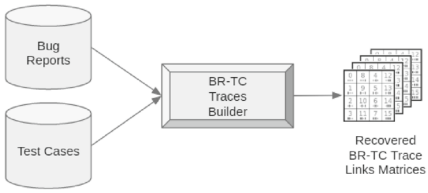


Fig. 12 *BR-TC traces builder* module

1. a word embedding with 300 dimensions and more than 1 million unique words is trained based on the Common Crawl Dataset;
2. tokens are extracted, without any preprocessing;
3. tokens are grouped in a matrix of word vectors representing each document; those vectors 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 its 300-dimension vector represents each one of them;
4. the average of grouped vectors is calculated for each document, generating smaller matrices;
5. cosine similarity is calculated for each paired test case-bug report.

Observe that the Word Vector technique correctly ranks the test cases, identifying test case 37 as the most relevant for the bug report, although the difference of the attribute similarity scores is not so precise, considering the scale (cosine similarity) between −1 and 1.

3 Approach

The case study applies IR and DL techniques as an *external/pluggable module*, in order to recover traceability links between bug reports (source artifacts) and test cases (target artifacts). During the analysis of Mozilla Firefox's data (Sect. 4), we used system features as intermediate artifacts, since most traceability links between bug reports and test cases cannot be recovered using only the information provided directly by the testers, as links between test cases and bug reports are not required during test-days by the Mozilla's leading teams. In particular, system features make the communication between the test and development teams easier, enforcing a common vocabulary. Analyzing the artifact organization, we noticed that if a bug report could be related to one specific feature, then it would be linked to the test cases of this feature. Figure 11 shows how these artifacts are

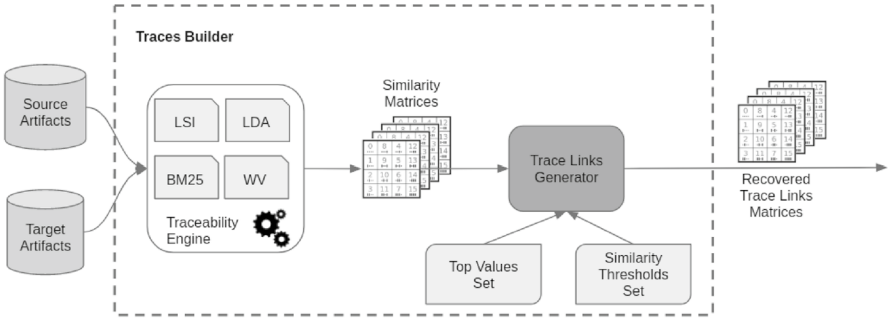


Fig. 13 Traces builder submodule

related to each other, where *BR_X* are bug reports, *Feat_Y* are system features, and *TC_YW* are test cases. We discuss this threat to validity in Sect. 7.

As seen in Fig. 12, 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 applied technique. Figure 13 schematizes a *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, for each applied technique, 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). In this 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 $d_j = (w_1, w_2, \ldots, w_N)$ and a query vector $q = (q_1, q_2, \ldots, q_N)$ as presented by Eq. 5 (Yadla et al. 2005; Buttcher et al. 2010).

$$sim(d_j, q) = cos(d_j, q) = \frac{\sum_{i=1}^{N} w_i . q_i}{\sqrt{\sum_{i=1}^{N} w_i^2 . \sum_{i=1}^{N} q_i^2}} \tag{5}$$

where $w_i = q_i = tf_i * idf_i$, tf_i is the frequency of a term i in a document (w_i) or query (q_i) and idf_i is the inverse document frequency of i. The *Trace Links Generator* receives three inputs: the set of similarity matrices generated by the *Traceability Engine*, a set of *Top Values*, and a set of *Similarity Thresholds*. A 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 (Antoniol et al. 2002; De Lucia et al. 2006; Dekhtyar et al. 2007; Guo et al. 2017).

Top Values define absolute values of documents to be recovered; for instance, TOP-1 only returns the document with the highest similarity score, whereas

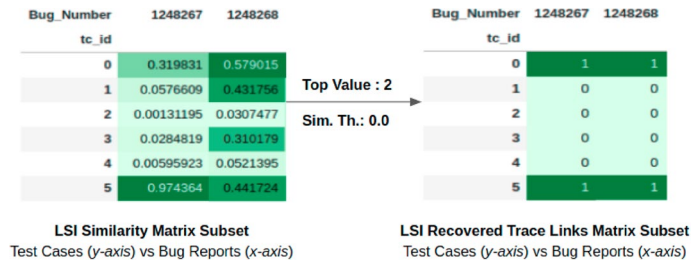


Fig. 14 Traces recovering process example

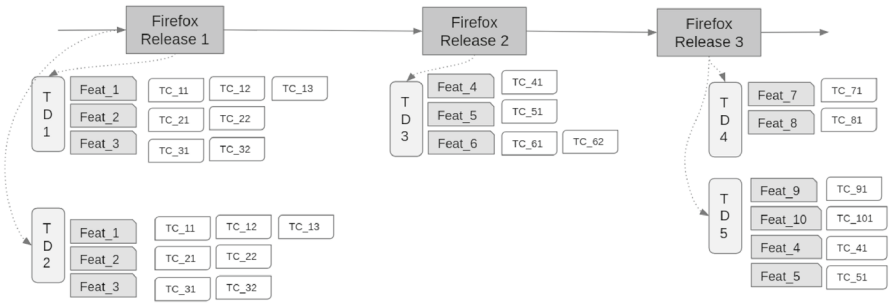


Fig. 15 Mozilla firefox's rapid release

TOP-3 returns the first three documents with the highest similarity scores. In addition, *Similarity thresholds* designate a minimum similarity score between a document and a query that must be reached by a technique. For example, (TOP-3, 0.85) states that only the three documents with the highest similarity values must be recovered, all of them with a similarity value higher than or equals to 0.85. Therefore, each similarity matrix cell will be set as a positive trace link (1) or not (0). Consequently, as we see in Fig. 13, for each combination of Top Value and Similarity Threshold an output matrix called *Recovered Trace Links Matrix* is created. Figure 14 shows an example of the recovering of trace links from the LSI's similarity matrix for (Top-2, 0.0) input. On the right side of the figure, you can note that only 2 test cases are returned for each bug report, corresponding to the highest similarity scores. The positive (returned) traces are depicted with the value 1, while the remaining ones are depicted with the value 0.

4 Building an oracle matrix

We describe in this section how the ground truth (oracle matrix) was created for the analysis in our case study (Sect. 5). The oracle matrix maps bug reports to Firefox's system features, with the help of a crowdsourcing application to gather the answers from volunteers and a researcher.

4.1 Context

The Mozilla Firefox internet browser[7] 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 (Mäntylä et al. 2013), in which they select a set of features for testing during a *test-day* at the end of each sprint (Each Firefox release has at least one test-day.) After all test cases for the features under test are executed, the set of bug reports is recorded. Figure 15 details the RR development model with three released versions of Mozilla Firefox, highlighting the test-days (*TDx*), the features tested in each test-day (*Feat_Y*) and the test cases of each feature (*TC_W*).

Core members of the Mozilla's QA team organize test-day data into an open-access Etherpad[8] 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 needed to be checked in that test-day. By the end of a test-day, 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[9] and create a link in the etherpad document as the result of the failed test case for later traceability.

However, testers often neither create the links as required nor create the bug report. 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

Since the task of building traceability links between bug reports and manual test cases requires in-depth knowledge about the system, recruiting volunteers fully engaged and immersed in this context would constitute a difficult and probably ineffective task. On the other hand, electing an expert and leaving the judgment and decision to build these links in his/her hands certainly could bring its cons, mainly decreasing the reliability of the generated oracle.

Therefore, in order to build the traceability links, we resort to the role of an expert, fully immersed in this task, with an adequate level of understanding of all the aforementioned artifacts. We completely rely on the expert answers (traceability links). However, to preserve the reliability of this task, we also recruited volunteers to add redundancy. The ground truth was produced by only considering links indicated by the expert that have also been indicated by the volunteers,

[7] https://www.mozilla.org
[8] https://public.etherpad-mozilla.org/
[9] http://bugzilla.mozilla.org

Table 1 Identified Firefox features

Feature name	Firefox Version	Number of TCs
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

i.e. the final oracle is constituted by the *intersection* between expert's and volunteers' answers. 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* 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 many more test cases (195) than features (19), and relating bug reports directly to the test cases would require a unfeasible amount of manual work.

A total of nine volunteers were recruited by e-mail invitation; they all have a Bachelor's 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 on the tasks, 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 – carried out the same tasks of volunteers, and another *matrix of traceability links* was generated from his answers.

As previously mentioned, the ground truth was produced by only considering links indicated as by the expert as by the volunteers, *i.e.* the final oracle is constituted by the intersection between expert's and volunteers' answers. All answers provided as by the expert as by the volunteers are available at the study website.

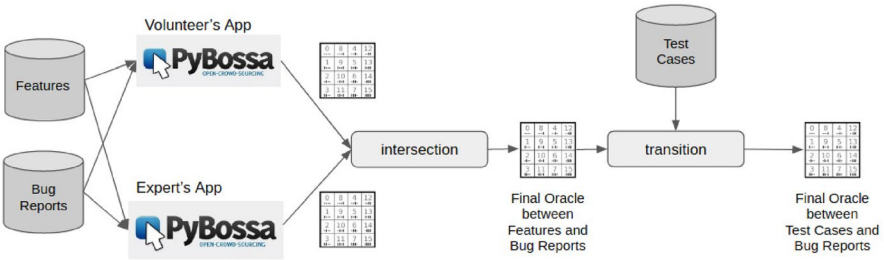


Fig. 16 Oracle creation process

4.3 Datasets

The used dataset of test cases and system features was extracted from Firefox test-days 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 test-days. 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 test-day available documents. Table 1 shows the Firefox Features used, the particular Firefox versions as well as the number of test cases associated with each feature.

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, which were the most up-to-date versions available at the time;
- Status must be RESOLVED or VERIFIED — other status levels potentially resulted in lower-quality bug reports;
- Priority must be P1, P2, or P3, the highest priority levels – considering the resources available for manual analysis, we chose to analyze the three most relevant types of bugs. We assume that bugs with lower priority could lead developers to be less concerned in organizing and writing a detailed report.;
- Resolution field must be FIXED, which means the bug was already fixed when collected for our study;
- Severity must be "major," "normal," "blocker," or "critical," ruling out "enhancements.", which tend to encompass bug reports left for subsequent sprints.

The Status[10] field indicates the current state of a bug. The Resolution field indicates if a bug was fixed or not. These filters reduced the number of bugs to be analyzed by

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

the volunteers in the study and they also allow selecting a subset of bug reports that are the most relevant in the entire data set.

4.4 Procedure

Our methodology was freely inspired by the one adopted in previous studies (De Lucia et al. 2006, 2009). The oracle creation was carried out following the process depicted in Fig. 16. As the input of the scheme, two datasets of System Features and Bug Reports feed the *PyBossa Platform*[11], which hosts the web applications to support firstly the participation of the expert, and secondly the participation of the volunteers. In a second moment, the intersection of volunteer's and expert's answers is produced generating a matrix of links between system features and bug reports. Finally, we derive by transitivity the matrix between test cases and bug reports (once we already possess the traces between test cases and system features) generating a final traceability matrix. For ground truth, we then consider each bug report manually associated with a feature linked to all manual test scripts for that feature.

After an extensive search, the Firefox team's text artifacts are the most complete documentation available for an open-source software process to the best of our knowledge. Its manual test scripts are explicitly linked to the artifacts they call features, although their bug reports do not point to the manual scripts that could reproduce the bug. Manually creating those links would be infeasible for experimental purposes as the number of manual test cases and bug reports combinations is considerable. Also, we did not have access to Firefox developers, and even if we had, they might not retain the information from artifacts dating from several months before. Our choice is inspired by a related work (Kaushik et al. 2011) which deals with the lack of previously-linked artifacts by using packages as a proxy for test cases, when mapping from bug reports.

We used the PyBossa crowdsourcing platform to coordinate the participation of each volunteer and aggregate his/her contributions; in this environment, it is defined an application or project which hosts a set of tasks. We created a set of 93 tasks, one for each bug report and two identical versions of these tasks were deployed to the volunteers' and the expert's applications. The workspace included the 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 of reading the bug report and the features descriptions, and thus decide which ones, if any, were related to that 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

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

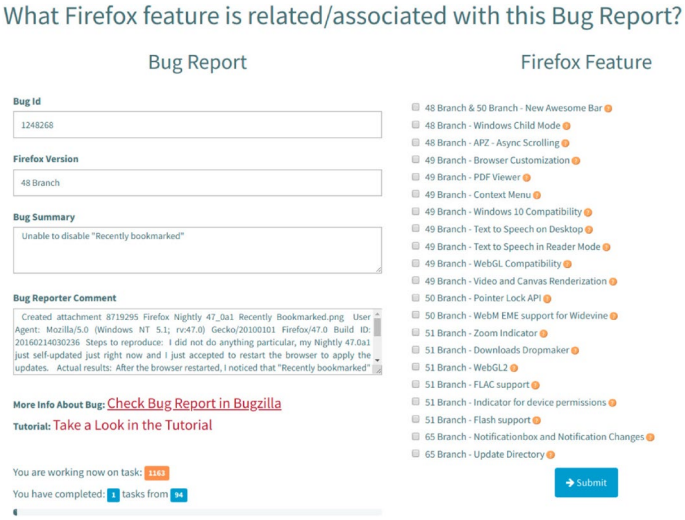


Fig. 17 Volunteers's application in PyBossa platform

Table 2 Number of traces in oracle grouped by system features

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

about the features[12], in case of the participants having doubts. Figure 17 shows a screenshot of the volunteers' application in the PyBossa platform.

All volunteers watched a 10-minute 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:

[12] https://support.mozilla.orghttps://wiki.mozilla.org/QA/https://www.paessler.com/manualshttps://addons.mozilla.orghttps://developer.mozilla.org

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

If a participant detected any of these conditions, he/she should indicate the existing (positive) relationship in the application's task submission, indicating thus an existing trace link between a bug report and a system feature.

4.5 Results

The volunteers' answers indicated the existence of 93 links between bug reports and system features, while the expert's answers indicated 58. Their intersection yielded 34 traces in total, resulting in a Kappa index of 0.46. By investigating the traces they do not agree on, some volunteers provided positive answers to visibly unrelated features; for instance, one bug report related to user data synchronization with the Firefox cloud system was linked, by more than one volunteer, to a scrolling feature. It is reasonable to assume the lack of expertise or experience is the main reason for differences in answers, by volunteers who are not part of Mozilla's development team. This threat is further discussed in Sect. 7.

The intersection oracle traces (positive links between bug reports and system features) are distributed as indicated in Table 2, where only seven features appear. Column *num_TCs* refers to the number of test cases from each Firefox feature, while *num_BRs* refers to the number of bug reports related to that feature as well as *num_Traces* refers to the number of traces (*num_BRs * num_TCs*).

5 Case study methodology

The case study that evaluates IR and DL techniques over Firefox data is reported in this section, using as input the links between bug reports and manual test cases produced as reported in the previous section.

5.1 Study definition

We aim to evaluate traceability between bug reports and test cases in the context of the Mozilla Firefox, using our approach as a basis. This study discusses answers to the following research questions:

RQ1 *Which is the most effective IR/DL technique?*

For the purpose of this evaluation, *effectiveness* is given by *Precision*, *Recall*, and F_2-*Score*. We used as baseline the Vector Space Model (VSM), which is broadly used in the field of Information Retrieval.

RQ2 *How does effectiveness vary based on variable cuts?*

This research question explores the impact of variations over combinations of similarity thresholds and top values over techniques' performance.

RQ3 *Which technique presents the best Goodness?*

The *Goodness* scale allows us to estimate the feasibility of a technique for application into a traceability recovery process.

RQ4 *Which is the best combination of cut values of each technique?*

This research question explores the multiple combinations of similarity thresholds and top values and allows us to estimate the best ones.

RQ5 *Which technique presents the lowest Recovery Effort Index (REI) coefficient?*

This question allows us to compare techniques in terms of effort saving in a traceability recovery process, from the perspective of a human analyst, in terms of the well-known REI coefficient.

RQ6 *Which technique presents the best run-time performance?*

This last research question allows us to explore the performance of each technique in relation to the time of traceability recovery, in seconds.

5.2 Context

The objects are bug reports and features from Firefox mapped in the full oracle, built in Sect. 4, along with test cases from which the features were extracted. As IR techniques, LSI, LDA, and BM25 were applied; also, two versions of Word Vector, as the DL techniques, based on different word embeddings. The values of *Similarity Thresholds* are in the range [0.0, 0.1,..., 0.9], considering each cut value based on the achieved similarity between bug reports and manual test cases. Also, we have used 10, 20 and 40 as *Top Values*, once the average number of test cases linked with bug reports is not larger than 40, varying the parameters of each technique accordingly. We used just three distinct Top Values for optimization purposes of the designed study, so it could be executed multiple times with the available resources.

5.3 Procedure

We did not use the system features content in the techniques application to establish trace links at this point, just the content of bug reports and test scripts. Each IR and DL technique and its preprocessing steps has its parameters defined according to the literature's recommendations. For preprocessing, we used Python's NLTK[13] (Natural Language Toolkit), a well-established framework for natural language processing applications, applying *tokenization*, *stop-word removal* and *stemming/lemmatization* to the artifacts' content. Additionally, the LSI's vectorizer uses *smoothing* as indicated in (Lucia et al. 2011, 2013) that its use can improve the traceability results. The same way we used in LDA's vectorizer. We also defined a maximum number of features in LSI (200 features) and in LDA (400 features) which tells the technique to only consider the top max features ordered by term frequency across the corpus, improving the performance in the traceability recovery process.

[13] NLTK: https://www.nltk.org

Next, we executed open-source implementations of the chosen IR and DL techniques – *Scikit-Learn Data Analysis Toolkit*[14] (LSI, LDA), the *Gensim Library*[15] (BM25), and the *SpaCy Library*[16] (Word Vectors). All scripts we implemented are available online[17]. SciKit Learn's LSI and LDA techniques require a vectorizer for manipulating tokens; the `TfidfVectorizer` implementation, provided by the framework, was used, along with NLTK's English stopwords. Furthermore, definitions for the number of components in dimensionality reduction (LSI) and the number of topics (LDA) are required; after testing several values, the best results for the algorithms were observed with 20 for both parameters.

Besides the number of topics (κ), the LDA requires others hyperparameters (Hoffman et al. 2012), namely the α, β and ν parameters. The α parameter influences the topic distribution per document, the β parameter influences the term's distribution per topic, while the ν parameter is the number of iterations required to the learning process to converge (Panichella et al. 2013). The α and β parameters we adopted is the SciKit-Learn framework's default value: $1/\kappa = 1/20$, such as the number of iterations which is $\nu = 100$.

In its turn, BM25's implementation was executed with the recommended values for English texts (Robertson and Zaragoza 2009; Canfora and Cerulo 2006); $k_1 = 1.2$ – the effect of term frequency – and $b = 0.75$ the effect of document length. Its 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 values into [0,1].

Finally, for the first version of Word Vector implementation, we used a pretrained neural network (word embedding) called GloVe (Global Vectors for Word Representation)[18] (Pennington et al. 2014) based on a vector space representation of more than 1 million tokens with 300 dimensions[19] extracted from blogs, news, and comments on the web in general (Common Crawl Dataset); for the second version, we used a word embedding trained only with the more than 35,000 bug reports collected from the Mozilla's Firefox bug tracking system, except the ones considered for the traceability recovery. We used two different word embeddings to evaluate the difference in the context of the texts for training the neural network may have over the traceability recovery capabilities in each version of Word Vector.

Following the preprocessing phase, the techniques are executed with tokenized bug reports and test cases. This process generated similarity matrices, yielding different *BR-TC Recovered Trace Links Matrices* according to multiple combinations of *top values* and *similarity thresholds*. As 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 of interest – the ones

[14] SciKit: https://scikit-learn.org/stable/
[15] Gensim: https://radimrehurek.com/gensim/
[16] SpaCy: https://spacy.io/
[17] https://github.com/guilhermemg/trace-links-tc-br
[18] https://nlp.stanford.edu/projects/glove/
[19] https://spacy.io/models/en

Table 3 Goodness level

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

with similarity greater than zero, meaning closer 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 Metrics

Precision, *Recall* and F_2-*Score* are very common metrics used in traceability recovery research (Hayes et al. 2006), and are defined in Eqs. 6, 7 and 8. *TP* denotes True Positives, *FP* False Positives and *FN* False Negatives. The F_β-*Score* is a general version of *F-Score*, and F_2-*Score* ($\beta = 2$) is an unbalanced version of F_1-*Score*. (F_1-*Score*) assigns equal importance to *Precision* and *Recall*, while (F_2-*Score*) assigns more importance to *Recall* over *Precision* (Berry 2017).

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

$$F_\beta\text{-}Score = (1 + \beta^2).\frac{Precision * Recall}{(\beta^2 * Precision) + Recall} \tag{8}$$

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 Participants Trace Links Matrix (Oracle)* 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.

The *Recovery Effort Index* (REI) was proposed by Antoniol et al. (2002) 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 a 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 thresholds employed in our study; in our version, we calculated REI for each technique and compared the obtained *Precision* with the *Precision* obtained by the oracle created only by the volunteers in relation to the oracle created by the expert. The REI value associated with a technique is the mean of all calculated REI's, as defined in Eq. 9. *OrcVolPrec* is the oracle *Precision*, $T_{i,j}Prec$ is the *Precision* of a technique with top

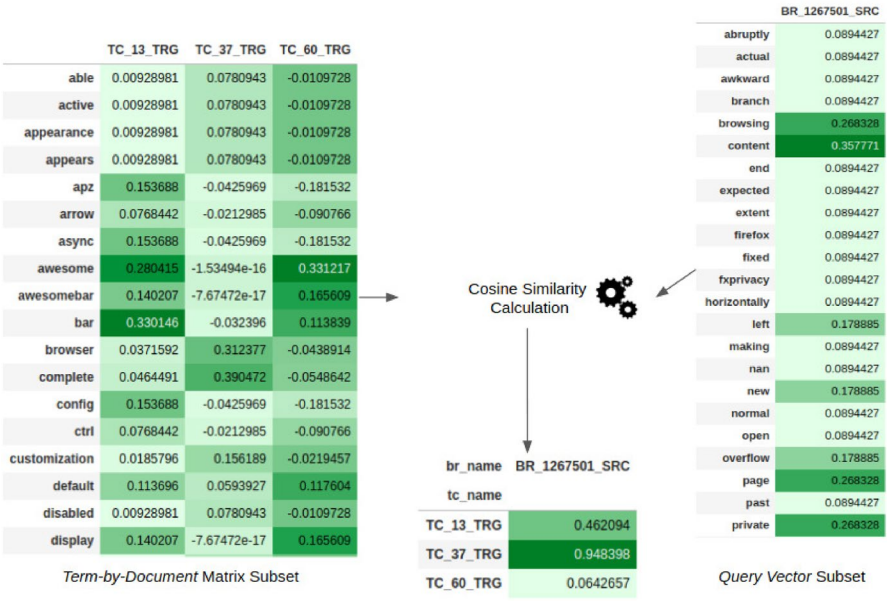


Fig. 18 Vector space model (VSM) example

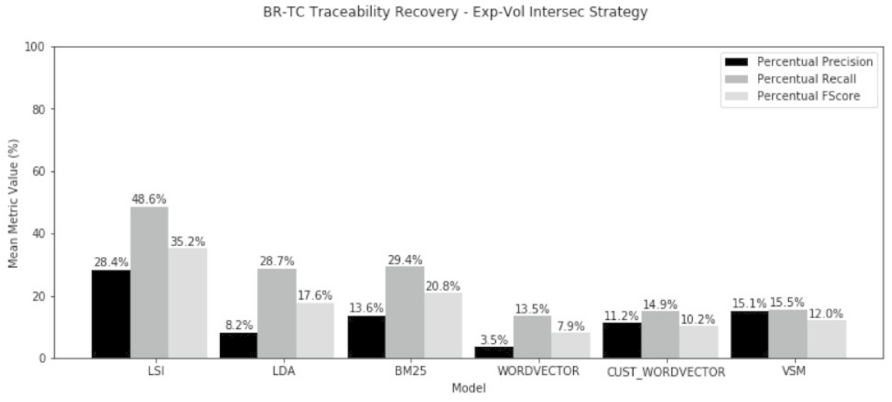


Fig. 19 BR-TC traceability recovery results

value i and similarity threshold j, and $|S_{i,j}|$ is the cardinality of the set of combinations of top values and similarity thresholds.

$$REI_T = \frac{\sum_{i,j} \frac{OrcVolPrec}{T_{i,j}Prec}}{|S_{i,j}|} \tag{9}$$

Additionally, we discuss the obtained results of *Precision* and *Recall* based on a scale of ***Goodness*** defined by Hayes et al. (2005), which establish some boundaries

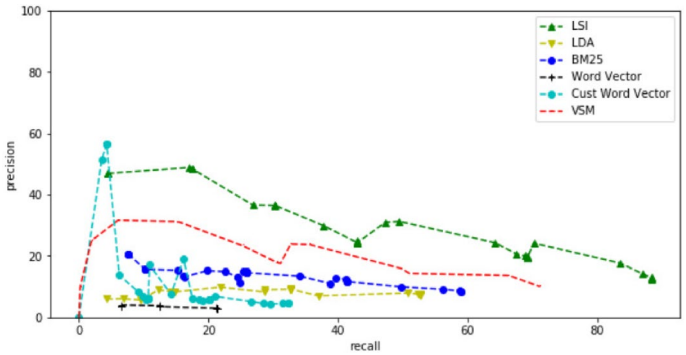


Fig. 20 PR-curves of all techniques—BR-TC context

for classifying the level of traceability recovery as Acceptable, Good or Excellent (Table 3). 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 Merten et al. (2016).

In order to have a baseline of comparison, we implemented a version of Vector Space Model (VSM) which is broadly used as baseline in field (Capobianco et al. 2009b, a; Oliveto et al. 2010) to classify a candidate trace between a bug report and test case as existent (1) or not existent (0).

The Vector Space Model (VSM) is one of the earliest and well-known models applied to Information Retrieval. It was developed by Gerald Salton in the 1970's and has been adapted to different applications like ranked retrieval, document clustering and classification (Salton et al. 1975; Buttcher et al. 2010). Many models and techniques were developed after the VSM model, such as the LSI model, which uses the same weighting scheme (Tf-IDF) and calculates the similarities between a document and a query using the same cosine function between the vectorized versions of these artifacts. The difference between the two techniques is essentially the Singular Value Decomposition operation applied in the LSI model. Figure 18 shows the VSM traceability recovery scheme. The bug reports and test case used in the example are the same used in Sect. 2 to explain the other techniques.

In our study we employ the same preprocessing operations used in the LSI technique, as described previously in the Subsect. 5.3. Also, we use the same implementation of Tf-IDF to vectorize and calculate the scores of each term in the documents (bug reports and test cases.

6 Results and discussion

We present and discuss the results from the study based on the research questions.

Table 4 Performance of techniques by top values

Model	Top	Precision	Recall	F_2-*Score*
LSI	10	30.89	34.39	**31.31**
LSI	20	28.67	50.77	**37.85**
LSI	40	25.72	60.72	**36.37**
LDA	10	14.19	20.27	18.09
LDA	20	13.89	37.02	26.19
LDA	40	11.53	46.98	25.61
BM25	10	15.58	20.41	18.61
BM25	20	13.69	29.76	22.08
BM25	40	11.67	38.03	21.72
WordVec	10	3.77	6.61	5.74
WordVec	20	3.55	12.45	8.29
WordVec	40	3.04	21.36	9.70
Cust. WV	10	11.06	8.15	7.52
Cust. WV	20	11.57	14.45	10.76
Cust. WV	40	11.00	22.21	12.25
VSM	10	15.66	10.97	10.47
VSM	20	15.18	15.71	12.62
VSM	40	14.50	19.68	12.91

```
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
```

Fig. 21 Topics generated by LDA technique

6.1 RQ1–Which is the most effective IR/DL technique?

Figure 19 presents the average of the results for precision, recall and F_2-*Score* for each applied IR and DL technique. LSI presented the best effectiveness for all metrics. Surprisingly, LDA 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 BM25 would achieve better performance. On the other hand, the Word Vector technique presented the poorest effectiveness

Table 5 Missed and captured traces: scenario I

Top	Missed traces	Captured Traces
10	196/514 = 38.13%	5/514 = 0.97%
20	99/514 = 19.26%	19/514 = 3.69%
40	31/514 = 6.03%	55/514 = 10.7%

concerning all metrics with *Precision* of only 3.5%, *Recall* of 13.5% and F_2-*Score* of 7.9%. The Customized Word Vector, trained with the own corpora of bug reports from the Mozilla, achieved slightly better results when we compare it with the first Word Vector implementation.

When analyze in the average, all techniques — except LSI — presented lower precision than the baseline, which presents precision of 15.1%. However, the recall of LDA (34.8%) e BM25 (29.4%) is higher than VSM's recall (15.5%). Analyzing through the use of F_2-*Score* metric, the best technique is LSI. Figure 20 depicts curves for all techniques, along with the reference curve of the VSM model (in red). LSI's values of precision and recall for almost every combination of top value and similarity threshold are the highest.

By aggregating the results by Top Value (10, 20, 40) (Table 4), LSI obtains higher precision and recall values for the highest Top Values when compared to VSM—this indicates the fixed cut is influencing results. Additionally, for Top 40, the LSI obtained in average an *Acceptable* level of *Goodness* (*Precision* > 20% and *Recall* > 60%), suggesting its feasibility for semi-automatic traceability recovery in projects such as the Mozilla Firefox.

The LDA technique was able to reproduce with trustworthiness the topics as system features, so the technique could split the test cases into groups that were very close to the features. Nevertheless, the technique was not able to achieve better results of *Precision* and *Recall* due to the low similarity characterized for those groups, and also due to some system features keywords that end up into the same topics. For example, bug report *1357458*, which refers to the *New Awesome Bar* feature, was correctly linked to the feature's 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, as illustrated by the highlighted tokens in Fig. 21.

Results from Word Vector were the lowest. The technique attributed high values of similarity for any pair bug report-test case, with a mean value of 0.91 and a standard deviation of 0.035. The technique was not able to capture the nuances between documents and assign diverse weights for the most relevant words in the text, not distinguishing relevant from irrelevant test cases for a given bug report. Therefore, new strategies still need to be elaborated for this kind of technique. As future work, we intend to explore variations of weighting schemes for specific targeted words in the vocabulary or make use of *enhancement strategies* (Borg et al. 2014) which better characterize the system features, so higher scoring values could be attributed to them. Also, strategies of preprocessing such as the one applied by Merten *et al.* (Merten et al. 2016) could be replicated into our context of traceability (see Sect. 8).

A recommended strategy to improve the Word Vector effectiveness is training a new word embedding with the texts from the context the model/technique is applied. This was made and the result is that, for the Customized Word Vector technique, the metrics improved. We noticed the mean similarity value in the Customized Word Vector technique is 0.41 and the standard deviation is 0.16, which means the technique did better in differentiating similarities in nearly all pairs of a bug report and test case. However, the Customized Word Vector was not successful in recovering the traces as the Word Vector, in a way that complementary strategies still need to be adopted.

6.2 RQ2–How does effectiveness vary based on variable cuts?

For this research question, we analyzed two different scenarios considering the range of similarity thresholds and top values. Due to the high number of combinations for the above parameters, we only select two scenarios to discuss: recall promoted over precision (similarity threshold as 0.0) and precision promoted over recall (similarity threshold as 0.9).

6.2.1 Scenario I: recall first

Table 5 shows the missed and captured traces on this scenario, for each top value. 31 traces (6.03%) were missed for the largest cut (40); they are related to three system features: *Context Menu*, *Downloads Dropmaker*, and *New Awesome Bar*, This phenomenon is also verified in the other Top Values, for which the missed traces are mostly related to the *New Awesome Bar* feature (nearly 51% for Top 10 and 63.3% for Top 20). These results are coherent with the number of traces related to these features in the oracle, as detailed previously in Table 2, where more than half of the traces (260 out of 514 or 50.5%) are linked to the *New Awesome Bar*, 93 (18%) to the *Context Menu*, and 72 (14%) to the *Downloads Dropmaker*. We estimate the larger number of missed traces is mainly due to the fixed cuts; the number of missed captured traces drops significantly with the increasing of the Top Value (only 6.03% in Top 40).

To better understand the missed traces, we analyzed six bug reports linked to those traces:

- BR_1276120 (New Awesome Bar): it presents no relevant keywords. The reporter used technical words that are distant to the vocabulary of test cases, such as “searchbar” and “urlbar”;
- BR_1279143 (New Awesome Bar): the description contains the word “awesomebar” written incorrectly;
- BR_1296366 (New Awesome Bar): the bug description is very brief and the title contains the word “awesomebar”, also written incorrectly;
- BR_1293308 (New Awesome Bar): contains technical words, such as “urlbar”, and a synonym “location bar”, both not used in the test cases descriptions;

Table 6 Traceability recovery results for scenario I

Top	Model	TP	FP	FN	Precision	Recall	FScore
10	BM25	133	778	381	14.6	25.88	22.41
	Customized Wordvector	55	855	459	6.04	10.7	9.27
	LDA	117	793	397	12.86	22.76	19.72
	LSI	221	689	293	24.29	43	37.26
	Wordvector	34	876	480	3.74	6.61	5.73
20	BM25	213	1609	301	11.69	41.44	27.46
	Customized Wordvector	103	1717	411	5.66	20.04	13.29
	LDA	222	1598	292	12.2	43.19	28.64
	LSI	356	1464	158	19.56	69.26	45.92
	Wordvector	64	1756	450	3.52	12.45	8.26
40	BM25	303	3339	211	8.32	58.95	26.59
	Customized Wordvector	166	3474	348	4.56	32.3	14.57
	LDA	302	3338	212	8.3	58.75	26.51
	LSI	454	3186	60	12.47	88.33	39.85
	Wordvector	110	3530	404	3.02	21.4	9.66

- BR_1270983 (Context Menu): this bug was probably reported automatically as a result of automatic test failure. Despite the presence of the word "contextmenu" (incorrectly written) in the title, the technique was not able to link it with the test cases of this system feature;
- 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 relative to this feature that have shorter descriptions.

Regarding the captured traces, less than 15% were captured by all techniques, even for the largest cut. $26/55 = 47.27\%$ of the traces linked with the features *Context Menu*, $11/55 = 20\%$ of *Indicator for Device Permissions*, and $8/55 = 14.54\%$ of *New Awesome Bar* were captured in Top 40.

Test cases related to system features such as *Context Menu*, *Indicator for Device Permissions*, and *New Awesome Bar*, present more words than the average, improving the inference to related bug reports. Additionally, their test cases commonly employ keywords that are very specific to the feature itself, often cited in the bug reports. In those cases, developers seemed to be more integrated, using similar vocabulary.

Also for Scenario I, True Positives (TP), False Positives (FP), and False Negatives (FN) brought up interesting discussion points. These results are depicted in Table 6.

The number of true positives of the LSI technique is considerably higher than the other techniques; for top 40, the technique was able to recover 88.33% of the

Table 7 Missed and captured traces: scenario II

Top	Missed traces	Captured traces
10	414/514 = 80.54%	0/514 = 0.0%
20	382/514 = 74.32%	0/514 = 0.0%
40	344/514 = 66.93%	0/514 = 0.0%

relevant links. The LSI correctly found 75 traces exclusively, which happened, for LDA, BM25 and WordVector, only 7, 6, 2 times, respectively. Through a qualitative analysis, we noticed LSI surpassed common difficulties, such as vocabulary differences between test cases and bug reports, as well as misspelled words, such as "awesomebar", which is referred to as "awesome bar" in the test cases. On the other hand, the other techniques were able to hit some hard-to-trace links. For example, LDA correctly identified a link between the bug report *1276120*, which has two "incorrect" words ("urlbar" and "searchbar") and no other indication from the related feature (New Awesome Bar).

LSI presented less False Positives (FP) than any other technique, while Word Vector is the first in FPs. Although the number of FPs is similar for all techniques – the number of FP grows identically with the increasing of the Top Value –, they incorrectly indicated traces relative to distinct system features. There are some illustrative examples of false positives by LSI, with top value 40. Bug report *"Show last sync date tooltip on Synced Tabs sidebar device names"* is not related to feature *Indicator for device permissions*. However, a link was reported to nearly every test case from this feature. Probably the technique was misguided by the presence of the word "device", understood differently in the test cases and bug report contexts. Also, bug report 1430603, briefly describing a technical issue involving implementation, presents blank recommended fields (steps to reproduce, expected results, etc.), is linked to test cases from eight features. We estimate this is due to the large size of the cut.

Regarding False Negatives (FNs), numbers are considerable. In a detailed analysis, we detected a large overlap among the techniques. For instance, LSI and BM25 yielded no exclusive false negatives in Top 10. By analyzing LDA's exclusive FNs, for top 40, we observe that 13 out of 21 of its omissions involved a single bug report, while 8 out of 26 involved a single test case. This bug report – *"Telemetry data from Search bar is not properly collected when searching in new tab from context menu"* – derives all FNs related to feature *Context Menu*. Our analysis shows this 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, it was difficult for LDA to trace links to Test Case *"Search State - Drop down"*, from feature *New Awesome Bar*. We believe the assigned topics were insufficient to grant a minimum similarity score between each of the eight bug reports and the test case, so the links could be traced into the top 40 cut. A probable cause for that maybe the longer text in this test case, if compared to other test cases from this feature.

Table 8 Traceability recovery results for scenario II

Top	Model	TP	FP	FN	Precision	Recall	FScore
10	BM25	39	150	475	20.63	7.59	8.69
	Customized Wordvector	0	1	514	0	0	0
	LDA	31	127	483	19.62	6.03	7
	LSI	23	26	491	46.94	4.47	5.46
	Wordvector	34	802	480	4.07	6.61	5.88
20	BM25	39	150	475	20.63	7.59	8.69
	Customized Wordvector	0	1	514	0	0	0
	LDA	37	159	477	18.88	7.2	8.21
	LSI	23	26	491	46.94	4.47	5.46
	Wordvector	64	1602	450	3.84	12.45	8.6
40	BM25	39	150	475	20.63	7.59	8.69
	Customized Wordvector	0	1	514	0	0	0
	LDA	37	168	477	18.05	7.2	8.18
	LSI	23	26	491	46.94	4.47	5.46
	Wordvector	109	3204	405	3.29	21.21	10.15

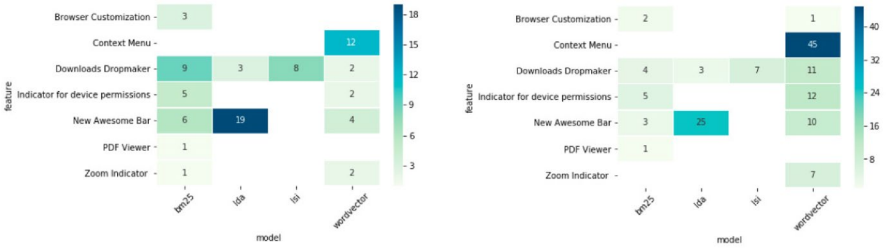


Fig. 22 Comparison of exclusive true positives – top 10 (left) and top 40 (right)

On the other hand, the Word Vector technique exclusively missed traces concerning all the seven relevant features, except *Context Menu*. However, the majority of missed traces is split between two features: *Downloads Dropmaker* (14) and *Indicator for device permissions* (11), being four bug reports for the former and two bug reports to the latter. Some of these bug reports may be considered easy to trace, such as bug report *"Search with default search engine stops working"*, which is correctly traced by all other techniques. The Word Vector technique seems not to be able to distinguish the relevant and irrelevant artifacts, even for major cut values. The algorithm to calculate the similarity between two documents is very naive and ignores the distinct weights the words may present.

BM25 uses the *bm25 weighting scheme* for estimating the weight of each word considering its source document and the entire corpus, achieving better *Recall*, having four exclusive FNs traces for top 40. All these traces were related to the *Context*

Menu system feature and originated from only two bug reports. One of them has also derived all exclusively missed traces of the LDA technique related to the *Context Menu* feature. This suggests this bug report may be especially hard to track, perhaps because it is related to two different system features. In this context, it is more difficult for the techniques to recover all the links.

6.2.2 Scenario II: precision first

Table 7 shows the missed and captured traces on Scenario II, for each top value. By setting the similarity threshold to 0.9, we enforce a high level of similarity for a reported link, so *Precision* scores tend to be higher than in Scenario I. As a consequence, there was an increase in the number of missed traces by the techniques; in general, test cases are made up of short texts, presenting insufficient words to grant high levels of similarity with bug reports.

Regarding the empty set of captured traces, 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 (Antoniol et al. 2002; De Lucia et al. 2006, 2009), despite the difference in the traced artifacts. The overall results for Scenario II are detailed in Table 8, with the selected top values. The overall results for Scenario II are detailed in Table 7, with the selected top values.

All techniques presented very low *Recall*, mostly below 10%. This is a critical issue since high *Recall* is a primary requirement for practical applications.

For the True Positives (TP), Customized Word Vector presented no true positives—zero—, while BM25 and Word Vector had the highest scores. For Top 10, BM25, LDA and Word Vector present a similar number of true positives; Word Vector was able to improve its performance with the increase in the Top Value, while the other techniques did not. The technique assigned high similarity scores to nearly all pairs of bug reports and test cases. The average similarity score is 0.907, with a standard deviation of 0.03, so it is expected that, as top values increase, the number of true positives also increases. On the other hand, Customized Word Vector, with average similarity score 0.41 and standard deviation 0.16, recovered no correct traces.

As long as 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, as shown in Fig. 22, where the traces are split by system feature (y-axis) and technique (x-axis). Note that LDA and BM25 "lose" traces from Top 10 (left-hand side) to Top 40 (right-hand side). Also, distinct system features are related to the recovered traces by each technique, indicating their inclination in detecting particular traces. For instance, in Top 10, BM25 had nine exclusive traces related to feature *Downloads Dropmaker*, while 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 indicate the potential value of a *technique combination* to address the traceability problem. This hybrid technique in such a scenario and with a Top Value of 10 would hit 78 traces out of the 514 possible, for example.

Table 9 Goodness scale for each technique

Model	Precision	Recall	FScore	Goodness
BM25	13.65	29.41	20.81	–
LSI	28.43	48.63	35.18	–
LDA	13.29	34.86	23.38	–
Wordvector	3.46	13.47	7.91	–
Customized Wordvector	11.22	14.94	10.18	–

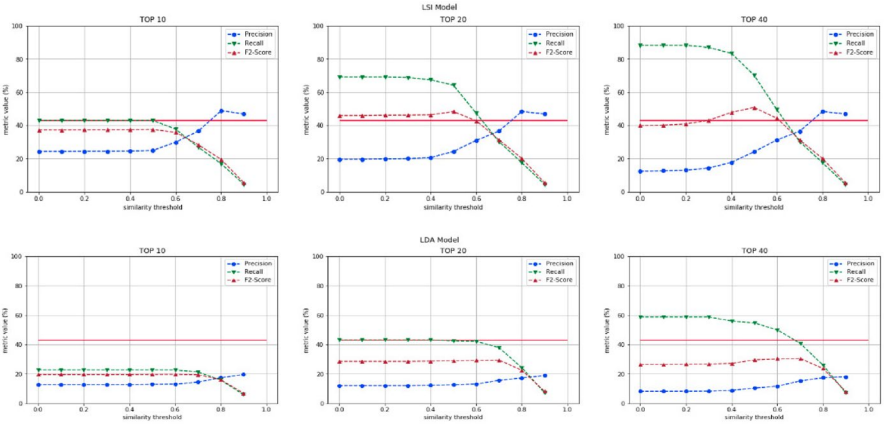


Fig. 23 LSI and LDA similarity threshold variation

Regarding False Positives (FP), while LSI had the lowest number of true positives, it presents the highest *Precision* scores. This number grows for every technique, except the LSI, which maintained the same 26 false positive recovered traces, independently of Top Value (Table 8). Word Vector tends to assign high values of similarity between the test cases and bug reports even if they are not related, leading to many FPs, increasing along with top values. For instance, the bug report *"Right click on bookmark item of 'Recently Bookmarked' should show regular places context menu"* is related with features *New Awesome Bar* and *Context Menu*, but the technique assigned high similarity scores (above 0.91) with feature *Windows Child Mode*. Differently from Scenario I, the FP results are not distinguishable for each technique, in terms of recovered system features.

For Scenario II, all techniques had poor False Negative (FN) results; *Recall* values were below 10%, except for Word Vector in Top 20 and 40, such as shown in Table 8. Also, all techniques had no exclusive false negative traces for any Top Value, which is mainly due to the fact the Customized Word Vector missed all traces. The results indicate the similarity threshold of 0.9 is not adequate for every technique, and an appropriate range must be carefully determined, in terms of the calculated results.

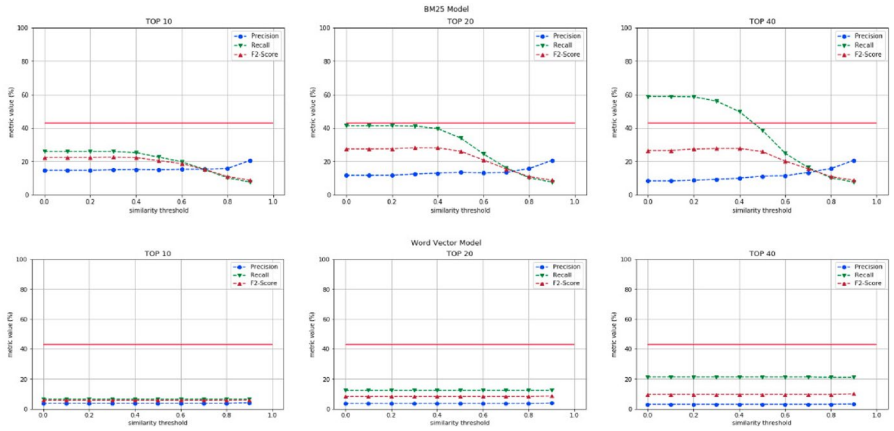


Fig. 24 BM25 and word vector similarity threshold variation

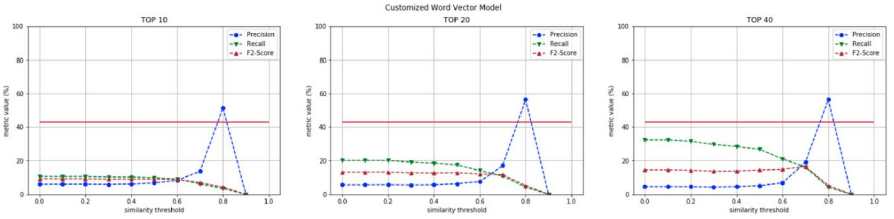


Fig. 25 Customized word vector similarity threshold variation

6.3 RQ3–Which technique presents the best goodness?

Adopting the *Goodness* scale, we calculated the levels of acceptance of each technique based on the values of *Precision* and *Recall* and the results are shown in Table 8. None of the studied techniques presented a satisfactory level of *Goodness* when we consider average *Precision* and *Recall*. Nevertheless, as explained in Sect. 6.4, 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 (Table 9).

The results indicate that LSI—using the identified best combination—is suitable for application in real and large projects such as the Mozilla Firefox. The human analysts or engineers can recognize the correct and incorrect traces between a pair of a bug report and test case, as well as to recover a considerable part of the trace links when using a traceability recovery tool with the LSI technique in their daily tasks.

Table 10 *REI* values

Model	REI
BM25	2.06
LSI	0.90
LDA	2.19
Word Vector	11.51
Customized Word Vector	–

Table 11 Performance times of each technique in seconds

Technique	Time(s)
LSI	2.65
LDA	3.28
BM25	2.29
Word Vector	42.11
Cust. Word Vector	96.8
VSM	1.59

6.4 RQ4–Which is the best combination of cut values of each technique?

In order to evaluate the hypothesis of existence of a best similarity threshold and to estimate it into the range of thresholds, we conducted an analysis whose results are shown in Figs. 23 and 24 and 25. Figure 23 depicts the effects of the variation of the similarity threshold in the LSI and LDA techniques, while Fig. 24 shows the effects over the BM25 and Word Vector, and Fig. 25 is relative to the Customized 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*. 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.

Analyzing the Figs. 23, 24, and 25, we can visualize a clear difference between the behavior of *Precision* and *Recall* scores in the evaluated techniques. In the IR 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 suffers 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.

When we look at the F_2-*Score* values, we see the most of them are below the minimum value of reference (red line). This value split *Acceptable* techniques from the not satisfactory ones. The F_2-*Score* of all techniques is 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%.

6.5 RQ5–Which technique presents the lowest recovery effort index (REI) coefficient?

The *Precision* of the volunteers' oracle (produced only by the volunteers) in relation to the expert's oracle - using it as a ground truth - is 42.66%. This score is used to calculate *REI* values. We summarize the obtained *REI* values in Table 10. Since *REI* is inversely proportional to *Precision* scores, and the LSI had the largest *Precision* in this study, it presented the lowest *REI*. The obtained results suggest LSI is the less time-consuming technique, regarding the analysis time required by traceability recovery tasks. LDA and BM25 require nearly as double time, whereas the Word Vector was eleven times slower. We did not calculate the *REI* coefficient for the Customized Word Vector once it is not defined for *Precision* values equal to zero, such as happens when the similarity threshold is 0.9. 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 we cannot attribute statistical significance to it without further study.

6.6 RQ6–Which technique presents the best run-time performance?

Table 11 shows the execution times of each technique, in seconds. We executed all techniques using an AMD Ryzen 5 3600 6-core processor machine, with 23GB RAM memory and Ubuntu 20.04 LTS operational system. The most efficient technique is the baseline VSM model, which took 1.59 seconds to recover the traceability links. Next, the BM25 model took 2.29 seconds to run, followed by the LSI model, with 2.65 seconds. As expected the VSM model took less time to recover the traceability links than the LSI model, since the latter is an improved version of the first, where the mathematical operation of Singular Value Decomposition (SDV) is applied to reduce the sparsity of the vectors used, although the operation requires an additional computational effort the final result is positive to the LSI model as discussed in the previous research questions. It is important to highlight the long time took by the Deep Learning techniques to recover the traceability links. This is a inherent phenomenon due to the huge amount of calculations made by the trained neural network to make a single inference, *i.e.* recover a trace link. A critical factor to this performance is the fact that the inferences were made in a CPU rather in a GPU as is very common in the use of such techniques. Additionally, they also have a bigger memory footprint, which is related to the size in memory of the trained models.

6.7 Discussion

In general, *Recall* levels from four out of five 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 seem to have satisfactory levels. These results suggest the terms used by bug reporters do not seem to match the terms used by Mozilla's QA team in test cases. Most bug reporters are not in Mozilla's test 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.

However, taking different combinations of top values and similarity thresholds, LSI presents an *Acceptable Goodness* level for some of them. The best 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 four techniques had poor effectiveness in any combination. Nevertheless, the techniques probably complement each other, at least in terms of true positives, which suggests better effectiveness using a hybrid technique.

Regarding the application of DL techniques, none of the variations applied achieved a significant result, neither Word Vector—which used a generic word embedding—nor Customized Word Vector—which used a word embedding trained with texts from the software engineering context; they are outperformed by all IR techniques for all metrics. Further studies must be conducted to prove the efficiency of this kind of technique for traceability tasks and involving bug reports and test cases as textual artifacts.

Also, there is still a considerable gap in the traceability recovery task for this type of traced artifacts. The results did not achieve *Excellent* for *Goodness*, either for *Precision*, *Recall*, or F_2-*Score*. The current level of *Goodness* grants an effective using of the LSI technique into a semi-automatized traceability recovery process, with human analysts or engineers working with the provided software tools for traceability recovery between bug reports and test cases. We estimate the effort required from the analyst using 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.

It is essential to highlight the relevance 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 to reach the automatic and precise linking with the bug reports, thus increasing the robustness of software development process and quality of the final product.

7 Threats to validity

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 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. 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 is that were used only the Firefox artifacts to draw the conclusions, which limits the generalization of the conclusions. We intend in future work to extend the approach to other systems so that we can claim more generality for the conclusions.

Errors of implementation not detected in the script used in the empirical study for data processing and analysis are 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. Furthermore, 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.

Manual test scripts, especially for agile teams, are considered a basis for requirements (Bjarnason et al. 2016). In those scenarios, no requirement descriptions are produced or maintained, except for user stories and backlogs, which are not detailed enough to encompass complex business upon which testers rely their validation efforts (Bjarnason et al. 2016; Sabev and Grigorova 2015) before sprint deliveries. In fact, despite the elevated cost of manual testing, automated system tests are rarely convenient due to the high rate of requirement change requests (Eder et al. 2014), as is the case of Firefox — in this case, regression system tests mostly follow manual scripts. In such a scenario, bug reports may not present a one-to-one correspondence with manual test scripts; those bugs often result from exploratory tests or automated unit/integration tests executed within a Continuous Integration (CI) context.

In this study, we argue that, although this formal link between bug reports and test cases is probably missing, the process could still benefit from a recommendation tool relating the bug report text with the text of a set of manual test scripts. By recommending this set to the testers, that tool would be essential to decrease the search scope for the required manual analysis of the bug's (indirect) impact on the system requirements. In particular, we agree with Hayes et al. (Hayes et al.

2006), who observed that IR techniques do not replace the human-decision maker when linking the artifacts. Still, instead, they should be used as candidate list generators. We believe our study could extend the knowledge on how traditional and state-of-the-art techniques support this approach to system tests. We assume, nonetheless, that a precisely built ground truth, including one-to-one mapping between those artifacts, would bring a more accurate account of the techniques' performance. We intend to keep a continuous effort to find publicly-available documentation from software teams, providing us with more complete artifacts and links.

Another problem that could be raised is related to the number of features shared between a bug report and a manual test case. In fact, in our study, a link between a bug report and a manual test case is traced whenever they share a system feature. So, it is possible to derive a sense of "strength" of a link between two artifacts based on the number of system features shared among them, but in our study we have not focused on this issue because there is no case of a bug report related to more than one system feature and each test case is exclusively related to a single system feature.

Despite the "strength" of the link between a bug report and a test case, we consider that it is worth to execute the entire suite of tests related to that feature if a bug report is linked and the code was changed during bug fixing - so the software quality is assured and new failures are detected before the new release. It is important to notice also that the number of tests (Sect. 4 Table 2) and complexity to execute them is not inhibitive based on the data we analyzed on in our study.

8 Related work

Comparisons between techniques in the traceability recovery context were carried out in previous studies. Falessi et al. (2010) 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 (2017) applies a set of popular machine learning models or 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. (2011) study traceability recovery for a private industrial system using *Precision*, *Recall*, and F_1-*Score* metrics. 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. (2016) 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. 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 verifies the superior effectiveness of the LSI technique over the BM25 when involving bug reports textual analysis. However, all techniques perform poorly 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). They also highlight the difficulties to track bug reports, such as the presence of *noise* in the text, such as hyperlinks, source code, stack traces, and repetitive information.

9 Conclusions

In this paper, we report a case study in automatically recovering traceability links between bug reports and manual test cases, through the use of system features to bridge the gap between those two types of artifacts. We compared the effectiveness of these techniques and observed the effectiveness of a traditional technique (LSI) in terms of well-known metrics. We have also assessed the applicability of a DL technique (Word Vector) for traceability recovery, which presented the poorest results. 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.

We observed that the usage 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. The best identified technique—LSI—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 test cases related to a determined bug report. Currently, the only requirements for using it is to provide manual test cases and bug reports in a textual format.

The present work has some limitations concerning the ground truth definition, the technique's parameters selected, and the statistical significance of the results. For the ground truth, 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 must be pursued. Additionally, a deeper parameter searching process could have been carried out, so the applied techniques would adopt the most adequate parameters for the software artifacts in the data set. 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.

The research work reported here opens a few investigation paths to follow in the future. 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. It is essential to consider, as well, the traceability of other software process artifacts, such as textual requirements, user stories, acceptance tests. It would thus provide evidence on the generalizability of the study's result.

Another path is the application of "enhancements" strategies with the LSI technique, such as building a thesaurus to deal with synonyms, clustering of documents/terms, phrasing, or query expansion techniques. Finally, another option is to evaluate the effectiveness of a hybrid technique created from the answers (returned traces) of the adopted techniques. Hopefully, the number of mistakes (false positives and false negatives) may be diminished if compared with the individual effectiveness of each technique. As we observed during this case study, the techniques tend to hit and miss different sets of traces, so that we suppose a combined version of them can compensate the failures of each technique individually, and also boost the number of correct traces recovered.

Acknowledgements We would like to thank the Brazilian agency CAPES for partially funding this research. We also would like to thank all the volunteers that kindly participated of our study. Last but not least, thanks for the anonymous reviewers whose contributions were decisive for the improving preliminary versions of the article.

Funding Brazilian federal agency CAPES of Ministry of Education (MEC/Brazil).

Data availability All the data is available in the following repository: https://github.com/guilhermemg/trace-links-tc-br.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability All the code is available in the same repository as the data and material: https://github.com/guilhermemg/trace-links-tc-br.

References

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002). https://doi.org/10.1109/TSE.2002.1041053

Berry, D.M.: Evaluation of tools for hairy requirements and software engineering tasks. In: Proceedings - 2017 IEEE 25th International Requirements Engineering Conference Workshops. REW **2017**, 284–291 (2017). https://doi.org/10.1109/REW.2017.25

Bjarnason, E., Unterkalmsteiner, M., Borg, M., Engström, E.: A multi-case study of agile requirements engineering and the use of test cases as requirements. Inf. Softw. Technol. (2016). https://doi.org/10.1016/j.infsof.2016.03.008

Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003). https://doi.org/10.1111/j.1365-2966.2012.21196.x. arXiv:1111.6189

Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empir. Softw. Eng. **19**(6), 1565–1616 (2014). https://doi.org/10.1007/s10664-013-9255-y

Buttcher, S., Clarke, C.L.A., Cormack, G.V.: Information retrieval-implementing and evaluating search engines. MIT Press, Cambridge (2010)

Canfora, G., Cerulo, L.: Fine grained indexing of software repositories to support impact analysis. Adv. Mater. Res. (2006). https://doi.org/10.4028/www.scientific.net/AMR.785-786.1516

Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., Panichella, S.: On the role of the nouns in IR-based traceability recovery. In: IEEE International Conference on Program Comprehension pp 148–157, (2009a) https://doi.org/10.1109/ICPC.2009.5090038

Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., Panichella, S.: Traceability recovery using numerical analysis. In: Proceedings - Working Conference on Reverse Engineering, WCRE pp 195–204, (2009b) https://doi.org/10.1109/WCRE.2009.14

Davies, S., Roper, M.: What's in a bug report?. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14 pp 1–10, (2014) https://doi.org/10.1145/2652524.2652541

De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: Can information retrieval techniques effectively support traceability link recovery?. In: IEEE International Conference on Program Comprehension **2006**, 307–316 (2006). https://doi.org/10.1109/ICPC.2006.15

De Lucia, A., Oliveto, R., Tortora, G.: Assessing IR-based traceability recovery tools through controlled experiments. Empir. Softw. Eng. **14**(1), 57–92 (2009). https://doi.org/10.1007/s10664-008-9090-8

Deerwester, S., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**(6), 391–407 (1990). https://doi.org/10.1017/CBO9781107415324.004

Dekhtyar, A., Fong, V.: RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow. In: Proceedings - 2017 IEEE 25th International Requirements Engineering Conference. RE **2017**, 484–489 (2017). https://doi.org/10.1109/RE.2017.26

Dekhtyar, A., Hayes, J.H., Sundaram, S., Holbrook, A., Dekhtyar, O.: Technique integration for requirements assessment. In: Proceedings - 15th IEEE International Requirements Engineering Conference. RE **2007**, 141–152 (2007). https://doi.org/10.1109/RE.2007.60

Eder, S., Hauptmann, B., Junker, M., Vaas, R., Prommer, K.H.: Selecting manual regression test cases automatically using trace link recovery and change coverage. In: Proceedings of the 9th International Workshop on Automation of Software Test, Association for Computing Machinery, New York, NY, USA, AST 2014, p. 29–35 (2014)

Falessi, D., Cantone, G., Canfora, G.: A comprehensive characterization of NLP techniques for identifying equivalent requirements. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10 p 1, (2010) https://doi.org/10.1145/1852786.1852810

Falessi, D., Di Penta, M., Canfora, G., Cantone, G.: Estimating the number of remaining links in traceability recovery. Empir. Softw. Eng. **22**(3), 996–1027 (2017). https://doi.org/10.1007/s10664-016-9460-6

Fazzini, M., Prammer, M., D'Amorim, M., Orso, A.: Automatically translating bug reports into test cases for mobile apps. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018 pp 141–152, (2018) https://doi.org/10.1145/3213846.3213869

Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. (2016) https://doi.org/10.1016/B978-0-12-801775-3.00001-9, arXiv:1011.1669v3

Gotel, O,C.Z., Finkelstein, A.C.W.: An Analysis of the Requirements Traceability Problem. In: 1st International Conference on Requirements Engineering (RE 1994) pp 94–101, (1994) https://doi.org/10.1109/ICRE.1994.292398

Guo, J., Cheng, J., Cleland-Huang, J.: Semantically Enhanced Software Traceability Using Deep Learning Techniques. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017 pp 3–14, (2017) https://doi.org/10.1109/ICSE.2017.9, arXiv:1804.02438

Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Tracing and mapping : supporting software quality predictions (2005)

Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans. Softw. Eng. **32**(1), 4–19 (2006). https://doi.org/10.1109/TSE.2006.3

Hayes, J.H., Dekhtyar, A., Sundaram, S.K., Holbrook, E.A., Vadlamudi, S., April, A.: Requirements tracing on target (RETRO): improving software maintenance through traceability recovery. Innov. Syst. Softw. Eng. **3**(3), 193–202 (2007). https://doi.org/10.1007/s11334-007-0024-1

Hemmati, H., Sharifi, F.: Investigating NLP-Based Approaches for Predicting Manual Test Case Failure. In: Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018 pp 309–319, (2018) https://doi.org/10.1109/ICST.2018.00038

Hoffman, M.D., Bach, F.R., Blei, D.M., Bach, F.R.: Online Learning for Latent Dirichlet Allocation. AcademiaEdu pp 1–5 (2012)

Kaushik, N., Tahvildari, L., Moore, M.: Reconstructing traceability between bugs and test cases: an experimental study. In: Proceedings - Working Conference on Reverse Engineering, WCRE pp 411–414, (2011) https://doi.org/10.1109/WCRE.2011.58

Kun, Chen, Wei, Zhang, Haiyan, Zhao, Hong, Mei: An approach to constructing feature models based on requirements clustering pp 31–40, (2005) https://doi.org/10.1109/re.2005.9

Lee, D.: 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

Lormans, M., Van Deursen, A.: Can LSI help reconstructing requirements traceability in design and test?. In: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR pp. 47–56, (2006) https://doi.org/10.1109/CSMR.2006.13

Lucia, A.D., Penta, M.D., Oliveto, R., Panichella, A., Panichella, S.: Improving IR-based traceability recovery using smoothing filters. In: IEEE International Conference on Program Comprehension pp. 21–30, (2011) https://doi.org/10.1109/ICPC.2011.34

Lucia, A.D., Di, M., Oliveto, R., Panichella, A., Panichella, S.: Applying a smoothing filter to improve IR-based traceability recovery processes?: an empirical investigation q. Inf. Softw. Technol. **55**, 741–754 (2013)

Manning, C.D., Raghavan, P., Schütze, H.: An introduction to information retrieval. Cambridge University Press, Cambridge (2009)

Mäntylä, M.V., Khomh, F., Adams, B., Engström, E., Petersen, K.: On rapid releases and software testing. Presented at the (2013). https://doi.org/10.1109/ICSM.2013.13

Merten, T., Krämer, D., Mager, B., Schell, P., Bürsner, S., Paech, B.: Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data? Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **9619**, 45–62 (2016). https://doi.org/10.1007/978-3-319-30282-9_4

Mikolov, T., Chen, K., Corrado, G., Dean, J. Efficient Estimation of Word Representations in Vector Space (2013) arXiv:1301.3781

Mills, C.: Automating traceability link recovery through classification. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017, ACM Press, New York, New York, USA, pp. 1068–1070, (2017) https://doi.org/10.1145/3106237.3121280

Minelli, R., Lanza, M.: Software analytics for mobile applications–insights lessons learned. In: 17th European Conference on Software Maintenance and Reengineering, pp. 144–153 (2013)

Oliveto, R., Gethers, M., Poshyvanyk, D., Lucia, A.D., De Lucia, A.: On the equivalence of information retrieval methods for automated traceability link recovery.In: IEEE International Conference on Program Comprehension pp 68–71, (2010) https://doi.org/10.1109/ICPC.2010.20

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshynanyk, D., De Lucia, A.: How to effectively use topic models for software engineering tasks? An approach based on Genetic Algorithms. In: Proceedings - International Conference on Software Engineering pp. 522–531, (2013) https://doi.org/10.1109/ICSE.2013.6606598

Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., Guo, J.: Feature-oriented software evolution p 1, (2013) https://doi.org/10.1145/2430502.2430526

Pennington, J., Socher, R., Manning, C.: Glove: Global Vectors for Word Representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Stroudsburg, PA, USA, pp. 1532–1543, (2014) https://doi.org/10.3115/v1/D14-1162, arXiv:1504.06654

Robertson, S., Zaragoza, H.: The Probabilistic Relevance Framework: BM25 and Beyond, vol 3. (2009) https://doi.org/10.1561/1500000019

Sabev, P., Grigorova, K.: Manual to automated testing: An effort-based approach for determining the priority of software test automation (2015)

Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11), 613–620 (1975). https://doi.org/10.1145/361219.361220

Sommerville, I.: Software engineering, 9th edn. Addison-Wesley, Boston (2010). https://doi.org/10.1111/j.1365-2362.2005.01463.x

Ng, A.Y., Jordan, M.: On discriminative vs. generative classifiers: a comparison of logistic regression and naive Bayes. Adv. Neural Inf. Process. Sys. **2**, 841–848 (2002)

Yadla, S., Hayes, J.H., Dekhtyar, A.: Tracing requirements to defect reports: an application of information retrieval techniques. Innov. Syst. Softw. Eng. **1**(2), 116–124 (2005). https://doi.org/10.1007/s11334-005-0011-3

Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., Weiss, C., Schröter, A., Weiss, C.: What makes a good bug report? IEEE Trans. Softw. Eng. **36**(5), 618–643 (2010). https://doi.org/10.1109/TSE.2010.63