JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **xx**:xx–xx
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Horizontal Traceability for Just-In-Time Requirements: The Case for Open Source Feature Requests

Petra Heck[1]* and Andy Zaidman[2]

[1]*Fontys Applied University, Eindhoven, The Netherlands*
[2]*Delft University of Technology, Delft, The Netherlands*

SUMMARY

Agile projects typically employ just-in-time requirements engineering and record their requirements (so-called feature requests) in an issue tracker. In open source projects we observed large networks of feature requests that are linked to each other. Both when trying to understand the current state of the system *and* to understand how a new feature request should be implemented, it is important to know and understand all these (tightly) related feature requests. However, we still lack tool support to visualize and navigate these networks of feature requests. A first step in this direction is to see whether we can identify additional links that are not made explicit in the feature requests, by measuring the text-based similarity with a *Vector Space Model* (VSM) using *Term Frequency - Inverse Document Frequency* (TF-IDF) as a weighting factor. We show that a high text-based similarity score is a good indication for related feature requests. This means that with a TF-IDF VSM we can establish horizontal traceability links, thereby providing new insights for users or developers exploring the feature request space. Copyright © 2014 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Traceability; TF-IDF; VSM; Information Retrieval (IR); Just-In-Time (JIT) Requirement, Feature Request; LSA

1. INTRODUCTION

Software evolution is an inevitable activity, as useful and successful software stimulates users to request new and improved features [1, 2]. A first step towards implementing these new features is to specify them. In projects developed using an Agile methodology this specification is typically informal, for example in the form of brief user stories which serve as conversation starters with stakeholders [3], or in the form of other 'software informalisms' [4].

This use of more lightweight representations for requirements contrasts the more traditional notion of requirements engineering (RE) and is also known as just-in-time requirements engineering [5]. Ernst and Murphy [5], Scacchi [4] and Mockus et al. [6] have previously observed just-in-time RE both in industrial projects and in open source projects. Mockus et al. even claimed

*Correspondence to: Petra Heck, Fontys Applied University, Eindhoven, The Netherlands. E-mail: p.heck@fontys.nl

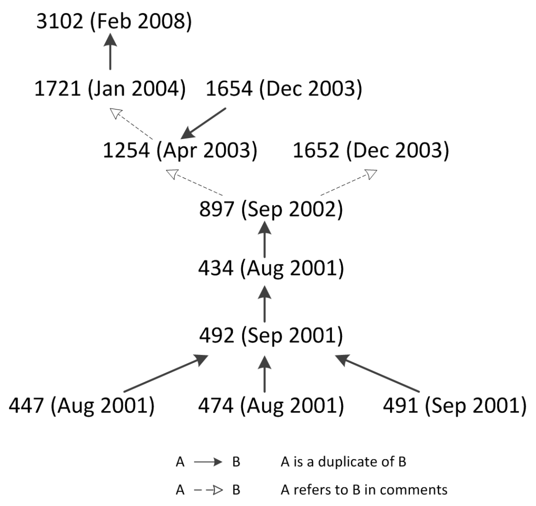

Figure 1. Feature request network in Subversion (feature request ID and creation date).

that despite the very substantial weakening of traditional ways of coordinating work, the results from open source software (OSS) development are often claimed to be equivalent, or even superior to software developed more traditionally [6].

Our previous research [7] has shown that just-in-time requirements engineering typically employs an issue tracker to record *feature requests*. On the one hand we have observed that this potentially leads to a large number of requests, which are sometimes difficult to search and navigate. On the other hand, we observed that large structures of relations exist between individual feature requests, see Figure 1 for an example of a so-called feature request network from the Subversion project [†].

Both when developers are trying to understand the current state of the system *and* trying to understand how a new feature request should be implemented, it is important to know and understand all these (tightly) related feature requests, to avoid duplicate or inconsistent development efforts [8]. For users reporting feature requests having knowledge of existing feature requests can also be important to know what is already being requested. In our previous research we observed that some of these relations are explicitly mentioned (textually) in feature requests [7]. In this paper we focus on automatically constructing horizontal traceability links [9] between feature requests, including the identification of additional links that are not made explicit in the feature requests, *e.g.*, because people are not aware of the related feature request. The more structure is provided, the easier it becomes for users or developers to follow these links and explore the related requests for the task at hand.

We focus on feature requests because as Cavalcanti et al. put it feature requests hold facts on the evolution of software, and thus can serve as documentation of the history of the project [10]. Discussions in some feature requests even show why the system has *not* evolved in a certain direction. Once a defect is closed ("fixed"), the information inside is not relevant for the current system anymore, because the defect does not exist anymore. However, once a feature request is closed ("fixed"), the information inside *is* relevant for the current system, because the feature still

[†] `http://subversion.apache.org`

exists in its original or evolved form. This means traceability between feature requests remains interesting, even for closed feature requests.

In order to establish traceability links we use a *Vector Space Model* (VSM) with a *Term Frequency - Inverse Document Frequency* (TF-IDF) weighting factor to calculate the text-based similarity of feature requests. In the remainder of this paper we will refer to the combination VSM - [TF-IDF] as "TF-IDF". TF-IDF has previously been applied to detect duplicate issue reports [11–14]. Issue reports include both defects and enhancements (feature requests). In our research we focus only on feature requests and are not interested in duplicates, but in requests that are functionally related.

We think that feature requests merit a separate text-based investigation with TF-IDF because we assume that the content of feature requests and defects is different, thus requiring a separate investigation. This assumption is supported by earlier work from Antoniol et al. [15] who devised a classifier for separating feature request and defects, and by Ko et al. [16] who used a decision tree algorithm to split feature requests from defects. An additional observation comes from Moreno et al. who claim that the terms used in bug reports are closely related to source code entities [17], which might not always be true for feature requests as they can be described more abstractly.

This leads us to the **main research question** for this paper: *Can TF-IDF help to detect horizontal traceability links for feature requests?*

Subsidiary research questions that steer our research are:

RQ1 Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked?

RQ2 What is the optimal pre-processing to apply TF-IDF focusing on feature requests?

Whereas TF-IDF only matches words that are literally the same, Latent Semantic Analysis (LSA) [18] takes into account words that are close in meaning by assuming that they will occur in similar pieces of text. However, LSA requires more calculation time because it adds additional calculation steps compared to TF-IDF. This situation intrigued us to see if the results for [RQ1] improve significantly enough when applying LSA to our application domain to merit this extra calculation time, which leads us to our final subsidiary research question:

RQ3 Does a more advanced technique like LSA (Latent Semantic Analysis) improve the detection of non-explicit links?

The remainder of this paper is structured as follows: Section 2 contains background with some major concepts and the related work. Section 3 explains our experimental setup. In Section 4 we describe the results of our experiment. In Section 5 we explore an example of a feature request network to illustrate our results. We discuss our results in Section 6. Section 7 concludes this paper.

2. BACKGROUND

In this section we discuss the background of our study. We start with a general background on traceability and then focus on horizontal requirements traceability. We further explain why we think the focus on feature requests (as opposed to issue reports also including defects) is necessary. We then discuss the use of TF-IDF/VSM for duplicate detection. We end by explaining both TF-IDF and LSA in a nutshell.

2.1. Traceability

"Traceability is the potential to relate data that is stored within artifacts of some kind, along with the ability to examine this relationship" [9]. Over the last 20 years or so, the software engineering research community has investigated numerous approaches to establish, detect and visualize traceability links [19].

An important branch of research has busied itself with so-called *requirements traceability* or "the ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [9, 20]". Requirements traceability can typically be described as a case of vertical traceability, where artifacts at different levels of abstraction are traced [9].

The investigation described in this paper however, can be catalogued as *horizontal traceability* as we are seeking to relate artifacts at the same level of abstraction.

De Lucia et al. have proposed to classify the traceability recovery methods according to the method adopted to derive links [21]: heuristic based, IR based, and data mining based. Recently, machine-learning methods have also been proposed to recover links between code and product-level requirements [22].

Many authors have successfully applied IR techniques including TF-IDF, e.g. [23–25]. Typical applications of IR include: concept location [26, 27], impact analysis [28], clone detection [29], software re-modularization [30] and establishing traceability links [23, 31–34]. See Binkley and Lawrie for an overview [35].

The investigation in this paper concerns recovering horizontal traceability links between artifacts of the same type, i.e. requirements.

2.2. Tracing requirements to requirements

We specifically discuss three works on tracing requirements to other requirements.

Huffman Hayes et al. [36] use IR techniques for tracing high-level requirements to low-level requirements. Candidate links generated by IR algorithms were to be confirmed by requirements analysts to measure performance of the algorithms. They extend TF-IDF with a simple thesaurus (manually made) to improve recall and precision of generated traceability links.

Natt och Dag et al. [37] use IR techniques for linking customer wishes to product requirements. They use a VSM with a different weighting factor [1 + 2 log (term frequency)].

Cleland-Huang [38] identifies three types of traceability that might be useful in agile projects. The third type is "requirements to requirements": to track dependencies between user stories. This can be useful during the planning process. She claims that there is no need to retain such traces once stories have been implemented. We would argue that this is different for feature requests in open source projects: the traces should be retained as they form an important part of project documentation.

We are not only interested in "traceability links" between feature requests, but also in related feature requests in general (e.g. about the same functionality). From our experience in open source projects there is not a habit of splitting high-level feature requests into low-level feature requests and we do not see any distinction in types of requirements, so with our TF-IDF analysis we can not make use of such an existing structure to find related requests.

2.2.1. Clustering Cleland-Huang et al. [11] propose an automatic clustering technique based on TF-IDF for the requirements-related messages on open forums. Their goal was to create threads of related messages. We focus on feature requests in issue trackers.

2.2.2. Networks Kulshreshtha et al. [39] studied literature about dependencies between system requirements and abstracted this into four dependency types: Contractual, Continuance, Compliance, Cooperation and Consequential. They modeled one such network from a set of 50 requirements pertaining to a Hotel Front Office Reservation system. Their study is on relationships between traditional requirements, where we focus on agile requirements (feature requests).

Sandusky et al. [40] studied what they call "Bug Report Networks (BRN)" in one open source project. This BRN is similar to what we have drawn in Figure 1. In the bug report repository they studied, 65% of the bug reports sampled are part of a BRN. They conclude that BRNs are a common and powerful means for structuring information and activity that have not yet been the subject of concerted research. The continuation of this stream of research will result in a more complete understanding of the contribution BRNs make to effective software problem management.

2.3. Feature Requests vs. Defects

As claimed in the introduction feature requests and defects have different characteristics. In this section we provide further support for that assumption from our own dataset and from existing literature.

The datasets we use are the feature requests as listed in the issue trackers of Mylyn Tasks, ArgoUML and Netbeans; more details on these projects can be found in Section 3. However, we start of with an investigation into who submits defects on the one hand and feature requests on the other hand. Our data is depicted in Table I. This table shows the total number of feature requests and defects, the number of submitters for feature requests and defects, and the total number of submitters. What we see is that the intersection of submitters of feature requests and submitters of defects is relatively small (between 13.5 and 21% of the total number of submitters)[‡]. This shows that the group of submitters for feature requests and the group of submitters for defects contain different persons. Those different persons might also be using different vocabulary.

Project	Feature requests		Defects		Submitters	Submitters
	#	# Submitters	#	# Submitters	total	feature req. $\bigcap$ defects
Mylyn Tasks	452	112	583	158	**223**	**47**
ArgoUML	1323	399	4847	1226	**1426**	**199**
Netbeans	29023	5017	204827	23071	**24737**	**3351**

Table I. Number of submitters for feature requests and defects

Herzig et al. investigated the relationship between issue reports and source code change sets. They found that this relationship is different for defects and features. Defect fixing change sets seem to change older code while feature implementations are based on newer code fragments. They also find that feature implementing change sets have more structural dependency parents in the change

[‡]It is known that sometimes the same person might submit under different user names, but we assume this holds only for a small number (possibly zero) of usernames in our large dataset.

genealogy graph than defect fixing ones. Above all defect fixing change sets show smaller impact on code complexity than feature implementations [41].

Herzig et al. [42] investigated the classification of existing bug reports (as indicated by the project members). They find that 34% of reports classified as "bug" is actually not a bug and only 3% of reports classified as feature requests is actually a "bug".

2.4. *TF-IDF and Duplicate Detection*

Several authors use Vector Space Models and/or TF-IDF similar weighting functions in the area of duplicate detection of bug reports [12–14, 43–45]. Runeson et al. [13] achieve a recall rate around 40% when analyzing defect reports using Natural Language Processing (NLP) with a vector-space-model and the cosine measure. Wang et al. [46] do not only use natural language processing but also use execution information to detect duplicates. Sun et al. [14] claim they obtain significantly better results (a relative improvement of the recall rates of 17-43% compared to the techniques used in [43], [13] and [46]) by using a discriminative model.

We only use duplicate detection to tune our pre-processing step. Our main research question is not focused on the best duplicate detection, but on the identification of horizontal traceability links for feature requests.

2.5. *TF-IDF in a Nutshell*

We use TF-IDF to convert a collection of documents into a collection of vectors (our Vector Space Model, VSM) by calculating the relative importance of each word in each document. The TF-IDF value increases with the number of times a word appears in a document, but decreases with the number of documents in which the word appears. This helps in filtering out words that are common in the entire collection of documents.

The formula that we have used for TF-IDF is as follows:

$$TFIDF(w,d,D) = TF(w,d) \times IDF(w,D) \tag{1}$$

with w a word in a document d that belongs to the collection of documents D. The Term Frequency $TF(w,d)$ can be computed in several ways. We choose to correct for longer documents, to prevent a bias towards longer documents:

$$TF(w,d) = \frac{f(w,d)}{max\{f(w,d) : w \in d\}} \tag{2}$$

with $f(w,d)$ the number of times that word w appears in document d.

$$IDF(w,D) = log_e \frac{|D|}{|\{d \in D : w \in d\}|} \tag{3}$$

with $|D|$ the total number of documents in the collection and $|\{d \in D : w \in d\}$ the number of documents where word w appears.

For a collection of documents D we first determine the complete set of unique words W. This set can be taken as is or can be normalized in many different ways (see section 3.1). Then we transform each document d that belongs to the collection D into a vector v_d. The dimension of this vector is equal to the size of the set of unique words W. The ith element of the vector v_d corresponds to ith word in the set W (w_i). For each d and i we calculate:

$$v_{d,i} = TFIDF(w_i, d, D) \tag{4}$$

Once we have each document vector we can calculate the so-called cosine similarity between two documents n and m.

$$\cos(v_n, v_m) = \frac{v_n \cdot v_m}{\parallel v_n \parallel \parallel v_m \parallel} \tag{5}$$

This yields a value between 0 and 1 indicating how similar the text of the two documents is: closer to 1 meaning more similar, 0 meaning no words in common at all.

2.6. *Latent Semantic Analysis (LSA) in a Nutshell*

Latent Semantic Analysis (LSA) [18] takes into account words that are close in meaning by assuming that they will occur in similar pieces of text. The input for LSA is a matrix. As described in [47] this is a weighted term-document matrix: the output matrix of the TF-IDF weighting algorithm as defined above (by combining all vectors v_d). This results in a matrix M of documents D against unique words W:

$$M[i,j] = TFIDF(w_i, d_j, D) \tag{6}$$

LSA uses a mathematical technique called singular value decomposition (SVD) to transform the so-called 'hyperspace' M into a more compact latent semantic space. This is done by maintaining the k largest singular values (also called rank-lowering of M). The optimal value of k is unique for each application domain. If the value of k is too low, the decomposition may end up under-representing the hyperspace M. Alternately, if the choice of k is too high, the decomposed sub-space may over-represent the hyperspace by adding in noise components [48].

After rank-lowering we have a new matrix M'. In this matrix M' each column represents a document vector. We calculate the so-called cosine similarity between two documents n and m in the same way as we do for TF-IDF above.

To summarize, the LSA technique adds reduced-rank SVD as an extra step compared to TF-IDF. It is believed that in the vector space of reduced dimensionality, the words referring to related concepts, i.e., words that co-occur, are collapsed into the same dimension. Latent semantic space is thus able to capture similarities that go beyond term similarity (e.g. synonymy) [48].

3. EXPERIMENTAL SETUP

In order to answer [RQ2] *What is the optimal pre-processing to apply TF-IDF focusing on feature requests*, we decide to make all pre-processing steps (see section 3.1) optional to find out which of these configurations yields the best results. We define 'the best results' as the configuration that succeeds best at finding the known set of duplicates. Known duplicates are feature requests marked as 'DUPLICATE' for the field *Resolution*. We use known duplicate feature requests from three mature open source projects. All three projects use Bugzilla[§] as an issue tracker to manage feature requests. We download these feature request as XML-files from the following three open source projects[¶]:

- Eclipse MyLyn Tasks `projects.eclipse.org/projects/mylyn.tasks`, 425 feature requests, 10 'Duplicate';
- Tigris ArgoUML `argouml.tigris.org`, 1273 feature requests, 101 'Duplicate';
- Netbeans `netbeans.org`, 4200 feature requests, 342 'Duplicate'.

We select these specific projects based on four criteria: 1) they are mature and still actively developed; 2) they differ in order of magnitude in terms of number of feature requests; 3) they have a (substantial) number of known duplicate feature requests; 4) they use Java as a programming language (important because some feature requests contain source code fragments).

We implement the TF-IDF calculation with possible preprocessing configurations in a C# tool, called FRequAT (Feature Request Analysis Tool)[‖]. FRequAT automatically verifies the similarity score against known duplicates (see Section 3.1). We use Mylyn Tasks and ArgoUML for the tuning of our algorithms and then confirm our findings with the Netbeans project.

Our FRequAT tool produces an Excel file that contains the pair-wise cosine similarity score for the complete set of feature requests. We do not have a golden set of all horizontal traceability links in the set of feature requests, thus we cannot use standard performance measures as recall and precision to evaluate TF-IDF for finding links. We therefor devise two different ways to answer [RQ1] *Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked*:
1) we manually analyze the top 50 most similar pairs of feature requests (*i.e.*, the pairs with the highest cosine similarity score). This manual analysis is done by the first author. For the pairs that do not have a physical link in the Bugzilla repository and that we think are functionally related, we ask for confirmation of our findings from one main developer from the Mylyn Tasks project and two main developers from the ArgoUML project.
2) we investigate an existing feature request network (one large example found during manual exploration of the repository of the Mylyn Tasks project) to validate that the existing links are supported by a high cosine similarity score and to see whether we can extend the existing network with additional links based on high cosine similarity scores.

To answer [RQ3] *Does a more advanced technique like LSA improve the detection of non-explicit links* we repeat the step of manually analyzing the top 50 most similar pairs of feature requests (see [RQ1] above) to see if we find more related requests with LSA than with TF-IDF only. We only do this for the Netbeans project because we deem the other two projects to small in terms of number of documents.

We explain some more details of the experimental setup in the following sections.

3.1. Preprocessing Feature Requests

The main step of the tokenization (*i.e.*, parsing the feature request text into separate words) is done through regular expressions. After that a number of configuration options are available to arrive at

[§]`www.bugzilla.org`
[¶]Datasets can be downloaded from `http://dx.doi.org/10.6084/m9.figshare.1030568`
[‖]Available from first author through email.

the final set of words to be considered for each document:

- *Include All Comments [AC].* The main parts of a feature request are: title, description and comments. This option configures if only the title and description or the complete feature request are used to build the VSM.
- *Set to Lowercase [LO].* This option sets all words to lower case.
- *Remove Source Code [SC].* This option uses extra regular expressions to remove different types of source code. The source code is completely removed, including comments, names of variables and identifiers.
- *Remove Spelling Errors [SP].* For this step we use a list of project-specific abbreviations and spelling mistakes that the first author manually compiled from the Mylyn Tasks project (we do not repeat this for the other projects because of their much larger vocabulary). If the option [SP] is on, each word is checked based on the Mylyn Tasks list and replaced by the alternative word before further processing.
- *Remove Stop Words [SR].* This option excludes known stop words from the VSM. We construct a list of stop words starting from the SMART list[**]. We manually add 66 stop words from the Mylyn Tasks project to this list (like 'afaik', 'p1', 'clr')[††]
- *Stem Words [SM].* This option reduces words to a common base: *e.g.*, 'browsing' and 'browse' become 'brows' and 'files' becomes 'file'. We use Porter's Stemming Algorithm [49].
- *Create Bi-Grams [BG].* This option builds the VSM based on sequences of two consecutive words, instead of based on single words.
- *Set Title to Double Weight [DW].* This option sets a double weight for the title as opposed to the description and comments [12, 13] .

Our FRequAT tool allows to switch the above options on or off. We define the best configuration as the one that yields the highest ranking for the known duplicates in a project. When we calculate the pair-wise similarity between a known duplicate and all other feature requests, then rank by similarity in descending order, the master of the known duplicate should be in the top of this ranking. A similar duplicate detection recall rate is proposed by Runeson et al. [13], who state that the standard information retrieval definitions of *recall* and *precision* do not really work for duplicate detection. The *Recall rate* is defined as the percentage of duplicates for which the master is found for a given top list size. We are interested in the TF-IDF configuration with the highest recall rate. In our evaluation we use top list sizes of 10 and 20.

We try several combinations of options on the Mylyn Tasks project and the ArgoUML project. We calculate the top-10 (T10) and top-20 (T20) recall rates for those combinations. Then we repeat the best combinations on the larger Netbeans project to get a confirmation of the highest recall rate found.

3.2. Most Similar Requests

We use the TF-IDF configuration with the highest recall rate for duplicate detection to see, for a given set of feature requests, whether we can get new information about related feature requests

[**]SMART list: `ftp://ftp.cs.cornell.edu/pub/smart/english.stop`
[††]Extra stop words `http://dx.doi.org/10.6084/m9.figshare.1030568`

by ranking them by pair-wise similarity. Our idea is that two feature requests with a high pair-wise similarity should be about the same topic and thus related.

The relatedness of two feature requests is determined in several ways:

1. The feature requests are related because they have one of the three existing link types in Bugzilla:
   - The 'blocks/depends' link. This is a manual link that is indicated by one of the project members, indicating that one should be resolved before the other.
   - The'duplicate' link. This is a manual link that is indicated by one of the project members.
   - The feature requests are related because one refers to the other in comments. This is manually done by one of the project members.
2. The feature requests are related because we as authors find them to be about the same topic.
3. The feature requests are related because one of the main project members considers them as related.

We calculate the pair-wise similarity for all feature requests from each project. Then we rank them and extract the top-50 most similar pairs of feature requests for each project. We choose 50 as a cut-off because 50 (times three for three projects) seems a reasonable amount to base conclusions on, and because we want to limit the effort for manual investigation. The top-50 is manually verified for the first three options above (existing links in Bugzilla) by the first author by looking at feature requests in Bugzilla. For the last two options we need a more extensive manual verification. The authors validate the remaining feature request pairs as being about the same topic or not. Lastly we send the remaining feature request pairs (meaning the ones in the top-50 not already linked in Bugzilla) to a prominent project member asking for his/her opinion about the relatedness. For this purpose we do not define 'relatedness', thus leaving it to the interpretation of the project member what he/she considers as 'related'.

3.3. LSA

We use the TF-IDF configuration with the highest recall rate for duplicate detection to further apply SVD (see Section 2.6). For application of the LSA algorithm we determine the k-value to use empirically by scanning the range (0 to 4200, i.e. the number of feature requests for Netbeans) in steps of 50. We use the k-value with the highest top-10 and top-20 recall rates for retrieving known duplicates. This is the same logic as described before to determine the optimal pre-processing steps for TF-IDF.

With this optimal k-value we calculate LSA-based cosine similarities and repeat the step of manually analyzing the top-50 most similar pairs of feature requests (see Section 3.2 above) to see if we find more related requests with LSA than with TF-IDF only.

4. RESULTS

This section describes the results of our experiments[‡‡]. We start off by describing the results of our investigation into the best pre-processing configuration for applying TF-IDF by using the known duplicate feature requests (Section 4.1). Subsequently, in Section 4.2 we analyze whether we can identify related feature requests, looking both for relationships that are already known and relationships that have previously not been documented explicitly. In Section 4.3 then, we compare the results that we have obtained with TF-IDF to the results obtained with LSA.

4.1. Best Pre-Processing Configuration

In order to establish the best pre-processing configuration(s), we present an overview of the different configurations and their recall rates in Table II. We observe that for the smaller Mylyn Tasks project several configurations score 90/100 recall rates, but when also taking the ArgoUML and Netbeans project into consideration, we observe that a single configuration scores best among all three projects.

In what follows, we discuss the influence of the pre-processing configurations.

	Mylyn Tasks		ArgoUML		Netbeans	
Options	T10	T20	T10	T20	T10	T20
BG	30	40	-	-	-	-
SR	50	50	58	71	40	49
None	**50**	**70**	**55**	**73**	**42**	**51**
LO	50	60	63	78	51	59
SP	50	70	55	73	42	51
SC	50	70	55	73	42	51
SM	40	70	57	74	42	53
DW	50	70	67	73	45	51
AC	80	90	60	72	57	65
AC_SC_SP	90	100	61	71	61	69
AC_DW_SC_SP	90	100	72	78	61	70
AC_DW_SC_SP_SM	90	100	73	79	60	71
AC_DW_SC_SP_LO	90	100	75	80	67	75
AC_DW_SC_SP_SM_LO	**90**	**100**	**79**	**85**	**67**	**77**
DW_SC_SP_SM_LO	50	80	75	83	55	63

Table II. Recall rates for the three projects

As Stop Word Removal (SR) is often performed as a standard step in research using TF-IDF [12] [43] [45], it is surprising to see that it yields worse results when applied. Apparently the built-in correction for common words (Inverse Document Frequency) in TF-IDF performs better than automatically removing all stop words from our manually compiled list. An option for future work would be to investigate the alternative technique presented by De Lucia et al. [50] to use a smoothing filter to remove "noise" from the textual corpus.

The Bi-gram option (BG) produced very low recall rates. The options DW, SC, SM, SP improve the recall rates.

[‡‡]Raw cosine similarity files as produced by our FReQuAT tool can be found on `http://dx.doi.org/10.6084/m9.figshare.1030568`

Regarding the All Comments (AC) option, from our first experiments with Mylyn Tasks, improved recall rates are achieved by including all comments. With the ArgoUML project the improvement is not clear. However, when adding Source Code Removal (SC) and Stemming (SM), the results for AC improve to the highest recall rates. This makes sense because when adding all comments we get a lot of diversity in the text of feature requests. Replies often include source code to provide a solution or hint for a solution. By removing the source code and stemming the words, the long feature requests get more similar. To be sure about this setting we repeat the experiment with AC off (last line in Table II). From this it is clear that overall we get better recall rates with All Comments.

To summarize, the best configuration (last bold line in Table II) is to activate all options except SR (stop word removal) and BG (bi-grams).

4.1.1. A Note on Categories of Duplicates In earlier work [7] we defined different duplicate types or categories: duplicate solution [DS], partial match [PM], different wording [WO], author knows [AU], no check done [NC], patch [PA], mismatch of attributes [MA]. In Table II it can be seen that with the best configuration 15% (ArgoUML) to 23% (Netbeans) of the duplicates is not detected within the top-20. We now analyze the duplicates that remain undetected within the top-20 of Table II to determine in which duplicate category they can be situated. It turns out that most non-detected duplicates are from three categories, see Table III:

- *Duplicate Solution [DS].* The two requests are not duplicate, but their solutions are. Developers tend to link feature requests as 'DUPLICATE' because they could be solved in a similar way. However the requests and their wordings are sometimes (completely) different. As such, text-based similarity sometimes does not detect this type of duplicates.
- *Partial Match [PM].* The master and the duplicate are not exactly the same because one contains more requests than the other. Only one of these requests is a duplicate of the other feature request. We then do not easily find this type of duplicates with text-based similarity analysis, because one of them contains extra text about an entirely different topic. An option for future work would be to investigate a different similarity measure such as the asymmetric Jaccard index to see if it better takes into account documents of different lengths.
- *Wording [WO].* The master and duplicate describe the same feature using entirely different terminology for the most important concepts. Examples we saw are 'nested class' vs. 'innerclass' or 'move' vs. 'drag/reposition'. Using TF-IDF we cannot detect such similarities without manual intervention. This motivated us to also investigate LSA as an alternative technique.

	ArgoUML	Netbeans	Netbeans LSA
Not detected in top-20	**15**	**78**	**74**
Partial Match [PM]	7	28	25
Wording [WO]	5	23	24
Duplicate Solution [DS]	-	17	15
No Check Done [NC]	3	8	8
Author [AU]	-	2	2

Table III. Categories of undetected duplicates. Categories taken from [7].

Two duplicates are of the category ‘Author’ [AU], meaning the author was aware of the duplicate. Normally we would hope to find duplicates from this category with TF-IDF because of use of similar words by the same author. In these 2 cases this does not work because of a long comment that is present in only one of the two requests, in one case, and because of the spelling of ‘outline view’ versus ‘outlineview’, in the other case.

The rest of the feature requests were of the category ‘No check done’ [NC], meaning it is not clear why the author did not detect the duplicate: searching for the main word in the title of the duplicate already results in finding the master. It is not immediately clear why using TF-IDF does not work well for these feature requests. It could be due to the fact that by using All Comments [AC] some of the feature requests become very long, increasing the chance that many other feature requests achieve a high similarity score. In that case, the real duplicate would not stand out any more.

Our analysis shows that out of the non-detected duplicates a number of them could not have been detected by (TF-IDF) text-based similarity and we can clarify that by the category they belong to. This means that a recall rate of 100% is not achievable for most projects (like ArgoUML and Netbeans in our study). Previous work on duplicate detection, e.g. [12–14, 43–45], did not investigate the duplicates that have been manually marked by the project. Our analysis shows that the data itself can be polluted (*e.g.*, the PM category are not real duplicates, although they have been marked as such), yielding lower recall rates than expected. This pollution of data could be solved by providing the project members with better linking mechanisms for feature requests, because now often the ‘DUPLICATE’ link is misused for other purposes (see also our previous work [7]).

4.2. Related Feature Requests

As explained in Section 3.2 we extract and rank the top-50 most similar pairs of feature requests per project.

Table IV summarizes the results of the top-50 analysis. The table shows, for each of the three projects, how many feature request pairs are physically linked (block/depend, duplicate, in comment) in the Bugzilla repository. This category of pairs is clearly related (we have evidence in Bugzilla) and does not need to be verified by the developers. As can be seen from the table, around 50 percent of the pairs is ‘linked’. The rest of the pairs needs to be considered manually. For the Mylyn Tasks project we contact the project leader and for the ArgoUML project we get feedback from two lead developers. We do not contact any developers for the Netbeans project because the manual verification by the authors does not lead to any doubt about the relatedness of the feature requests. The 50th most similar pair for the Netbeans project still has a similarity score just under 0.70, which is higher than for the first 2 projects (0.46 and 0.48). The higher the similarity score, the ‘closer’ the text is, the easier it is to determine relatedness. In Table IV we mark a pair as ‘Related, confirmed’ when both we and the project developers have indicated them as being related. As can be seen from the table a low percentage of pairs is not linked at all, around 10%.

We analyze why some of the feature requests are considered as related by the authors, but not by the developers. This mainly happens because of the difference in interpretation of ‘related’. For the authors ‘related’ means ‘on the same topic’ and for the Mylyn developer it means ‘in the same module of the software’. The developer definition should be investigated differently, *e.g.*, by analyzing commit logs. We stick to our definition of related because we think it is useful to have a link between feature requests that are about the same topic or feature. This can help, *e.g.*, new users

	Mylyn Tasks	ArgoUML	Netbeans	Netbeans LSA
Linked	20 (40%)	28 (56%)	29 (58%)	25 (50%)
Related, confirmed	15 (30%)	19 (38%)	-	-
Related, not confirmed	8 (16%)	1 (2%)	15 (30%)	17 (34%)
Not related	7 (14%)	2 (4%)	6 (12%)	8 (16%)
Unmarked duplicates	-	-	5	4

Table IV. Related pairs from top-50 most similar.

that would like to ask for an extension to an existing feature but first want to see what is already there, or it could help developers that are new to the project and would like to get an overview of which functionality is there from a user point of view. With our topic-based definition of relatedness we have around 90% related feature request pairs in the top-50.

For the Netbeans project we detect in the top-50 five pairs of feature requests that are indeed duplicates of each other. It is clear to us from the title and description that the two are real duplicate requests asking for the same feature. But those pairs have not been marked as such in the Bugzilla repository. These feature request pairs are called ‘Unmarked duplicates’ in Table IV. It shows that our method can also find new duplicate requests, i.e. not having a ‘duplicate’ link in Bugzilla yet.

An interesting side-effect of analyzing the top-50 most similar feature request pairs is that we see that the same feature request appears in several pairs in the top-50. This means that we can start to see small groups of related feature requests. Intuitively, these small groups should contain feature requests about the same topic. This is an example of such a group from the Mylyn project:

[102854] add bug editor support for **voting**

[156742] Add search criteria for tasks/bugs having **votes**

[256530] [upstream] **voting** for a bug capability missing from editor when no votes present

[316881] provide **voting** API in BugzillaClient

The Mylyn group about “Voting” can only be found completely when searching for “vot”. This search for the stem of a verb is not natural to an end-user and he/she risks not to find the complete set. That is why our TF-IDF approach can add value, even if it is just text-based on literal strings.

We also see examples of feature requests that are related according to TF-IDF scores (0.64) and according to the developer, but that do not have related titles, e.g.

[Mylyn 283200] [Bugzilla] Support querying over custom fields

[Mylyn 339791] Bugzilla search page did not store chart settings

A simple search for ‘Bugzilla’ in the Mylyn Tasks feature database yields 40 results among which the two related ones are hard to find based on the rest of the title. This example shows that the VSM with TF-IDF can find related feature requests that are hard or impossible to find with simple search in the issue tracker.

What we can conclude from this experiment is that high values of cosine similarity can be a good indication of relatedness between two feature requests. We will explore this further in Section 5 when looking at a feature request network.

4.3. LSA

With the same pre-processing as for the TF-IDF application (AC_DW_SC_SP_SM_LO) we determine the optimal k-value to be 1700. The top-10/top-20 recall rates for k=1700 are only slightly higher than for TF-IDF: 69/78 instead of 67/77. This could be due to the fact that our dataset is relatively small (only 4200 ‘documents’); when considering applications of LSA, most applications use 10,000 documents or more (e.g., see [18]). Nevertheless, we consider Netbeans with its 4200 documents as a large project. Another possible explanation of why LSA does not outperform TF-IDF is that the community of users of issue trackers use the same kind of terms to refer to concepts, whereas LSA would be better at linking different terms that have a similar meaning. This assumption needs to be checked in future work.

If we look at the categories of non-detected duplicates, we even find one more non-detected duplicate of the category ‘Wording’ [WO], see Table III. Furthermore we still do not detect any extra duplicates of the ‘No check done’ [NC] or ‘Author’ [AU] categories. This means LSA in our case does not perform better than TF-IDF for detecting those categories of duplicates, contrary to the expectation that we expressed in Section 4.1.1.

In the analysis of the top-50 most similar feature request pairs, we find many of the feature request pairs that were also there with TF-IDF (36 out of 50 are the same, although their position within the top-50 differs for some of them). Noteworthy is that the 50th most similar pair has a significantly higher cosine similarity than with TF-IDF: 0.79 instead of 0.69. When we analyze the top-50 most similar feature request pairs (see Table IV) the surprising result is that when we compare the results of TF-IDF with those obtained with LSA we see that for TF-IDF 88% of the top-50 feature request pairs are related (coming from the categories Linked and Related), compared to 84% in the case of LSA. Additionally, the TF-IDF result-set contains 6 unrelated feature requests, while for LSA this is slightly higher at 8. This indicates that TF-IDF results are slightly better for our dataset.

We do not invest time in optimizing the performance for both TF-IDF and LSA. Currently the processing time for calculating the output term/document matrix for the Netbeans project is approximately 3 minutes with TF-IDF and 900 minutes with LSA. This shows us that even with optimization the processing time for LSA would likely still be considerably higher.

5. EXTENDING A FEATURE REQUEST NETWORK

Knowing that VSM with TF-IDF can find new related feature requests, we want to look into what this means for the feature request networks as presented in the introduction. One would assume that the feature requests within such a network have a high pair-wise similarity.

Figure 2 shows a feature request network from the Mylyn Tasks project. Each node is a feature request identified by a six-digit unique ID. There are three types of links between the feature requests: 1) ‘blocks/depends’, 2) ‘duplicate’ and 3) comments. These three types are also explained in Section 3.2. For each link in the figure the pair-wise cosine similarity is indicated. If the feature request is linked to a defect, the cosine similarity is not calculated, indicated with a ‘D’. Indeed we can see that this similarity ranges from 0.33 to 0.69 with a few low outliers. Our explanation for those low numbers:

- *0.04/0.06/0.09*: this is due to a partial match (see also our previous work [7]). Feature request [354023] is about the implementation of a web service between Mylyn Tasks and Bugzilla. The other 3 feature requests linked to [354023] need this web service to be implemented (hence the links), but are not really about the same topic.

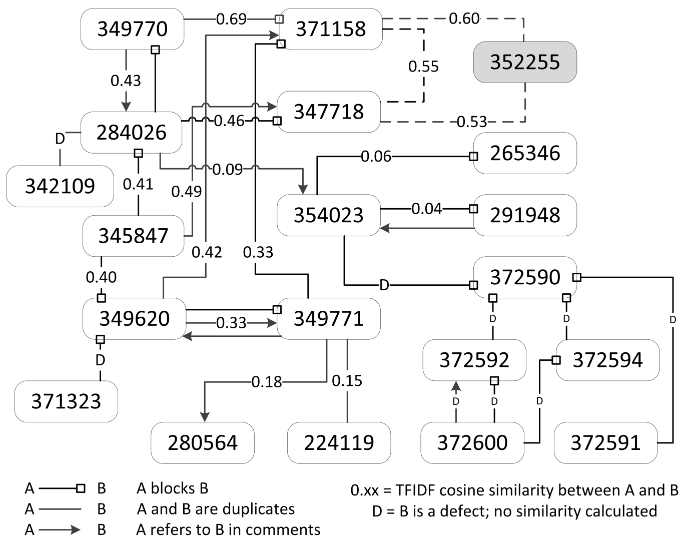

Figure 2. Feature request network for the Mylyn Tasks project

- *0.15*: feature request [224119] describes a new functionality in Bugzilla that needs to be available in Mylyn Tasks from the Bugzilla viewpoint. Feature request [349771] describes the same, but from a Mylyn viewpoint with much more detailed discussion. That is why the two feature requests are not so similar. However, they are known duplicates. In our duplicate analysis the duplicate is still found within the top-20 because other feature requests are even less similar to [224119].
- *0.18*: again a partial match. Feature request [280564] is about creating scalable icons in general. Feature request [349771] is about the lock icon, that needs to be scalable as well, but the scalability is just one of the many comments in this feature request. However, if we rank the most similar feature requests for [280564] then [349771] is on the 7th place, thus within the top-10 most similar.

This analysis teaches us that we should not only look at absolute similarity numbers to find related requests, but more specifically include the relative similarity. This could be done by, *e.g.*, taking into account the mean m and standard-deviation s of the pair-wise similarity of feature request A with all other feature requests, when calculating the relative similarity between feature requests A and B.

In Figure 2 we also add feature request [352255] with dotted lines. This feature request is in the top-50 of highest pair-wise similarity for the Mylyn Tasks project. In this top-50 it is linked to two other feature requests, [371158] and [347718], with a high cosine similarity. Interesting to see is that [347718] and [371158] also have a high pair-wise similarity while they have no direct link in the issue tracker. We are thus able to extend the physical feature request network in the issue tracker with a new feature request and new links through our TF-IDF calculations.

6. DISCUSSION

6.1. *Revisiting the Research Questions*

In the introduction we set out to investigate whether TF-IDF can help to detect horizontal traceability links for feature requests. Answering this step is part of our bigger research ambition to visualize related feature requests that are stored in issue trackers. Before answering this high-level research question, let us first consider the subsidiary research questions.

RQ1 *Is TF-IDF able to detect functionally related feature requests that are not already explicitly linked?* We show that a VSM with TF-IDF can be beneficial to detect new related feature requests. We cross-check our results with feature requests that are already marked as related in the issue tracker, with our own opinion while reading the feature requests, and for two out of the three projects (Mylyn Tasks and ArgoUML) also with the help of experienced project members. We confirm this by means of an existing feature request network from the Mylyn Tasks project. This analysis teaches us that we should not only look at absolute similarity numbers to find related requests, but more specifically include the relative similarity.

RQ2 *What is the optimal pre-processing to apply TF-IDF focusing on feature requests?* Through case studies with the Mylyn Tasks, ArgoUML and Netbeans projects we determined the optimal pre-processing does not include stop word removal, while removal of source code is beneficial. Additionally, we determined that all comments of the feature request should be included. We attribute this to the fact that important words concerning the feature are repeated in the comments.

RQ3 *Does a more advanced technique like LSA (Latent Semantic Analysis) improve the detection of non-explicit links?* The results of our experiment with LSA show us that our initial assumption (that LSA would improve results as it takes into account e.g. synonyms) does not hold for the Netbeans case. We are not interested in achieving higher recall rates, but in finding related requests. LSA scores lower on finding related request based on the top-50 analysis. This makes us even more satisfied with the results achieved with TF-IDF (also because the LSA calculations take much more time).

Main research question *Can TF-IDF help to detect horizontal traceability links for feature requests?* We can confirm TF-IDF finds related entries in an issue tracker. In our experiment, on the one hand we retrieved already known relations between feature requests, while on the other hand we retrieved feature requests pairs that were previously not marked as related, but that are confirmed to be related by project members after we confront them with our results. We explain our results for one example of a feature request network.

6.2. *Threats to Validity*

We now identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [51, 52]) we organize them into categories:

Construct validity We evaluate the best configuration of our text-based similarity approach via the recall rate of previously known duplicate feature requests. These duplicates were marked by the project team. Similarly, for evaluating the related feature requests, we rely on information in the issue tracker entered by the project team, on our own opinion and on extra insights obtained from project members. What must be noted in this case is that for the ArgoUML project the two different developers do not always agree on related or not related (discussing between 'yes' and 'somewhat'), although they only completely disagree on 1 item (out of 14). As one of the developers says this difference is because "it is subjective whether two feature request are related" (*e.g.*, related in terms of implementation or in terms of topic). This could mean that if we would ask different developers we would get different answers.

External validity In this study we investigate feature requests of three software projects: Mylyn Tasks, ArgoUML and Netbeans. We choose them to be sufficiently different, in terms of domain and size. Yet, with only three data points, we cannot claim that our results generalize to other systems.

Reliability In this paper we rely on our FRequAT tool, which we have thoroughly tested and which we consider reliable.

7. CONCLUSION

This paper describes three case studies with feature requests from open source projects. Our main contributions are:

1. TF-IDF can be used to detect horizontal traceability links for feature requests, something which we validated for 2 out of the 3 projects with the help of developers.
2. Configuration of the pre-processing step is analyzed separately, because we focus on feature requests. For our three projects stop word removal is not beneficial, whereas including 'All Comments' and removing source code is yielding better recall rates.
3. LSA does not provide better results than TF-IDF in detecting horizontal traceability links for feature requests in our case studies.
4. When using 'Duplicate' links in issue trackers to base recall rates on, one should be aware of data pollution caused by misuse of the 'Duplicate' link, yielding lower recall rates.

Our results will help others to properly set up pre-processing for information retrieval techniques with feature requests, and to get insight in feature request networks. Especially those feature request networks play an important role in understanding the evolution of the specification of the system (recorded in the form of feature requests).

The results in this paper show that TF-IDF can help to detect horizontal traceability links for feature requests. A next step is to get measures on thresholds, recall and precision for the retrieval of those links, by using (industrial) case studies where we are able to identify all those links on before hand.

A subsequent step is to create tool support for automatically creating feature request networks so that users can maximally benefit from the horizontal traceability links.

Another avenue for future research is to compare additional information retrieval approaches for our particular problem domain, in similar vein to Oliveto et al. [53]. Part of that investigation should also determine why LSA is currently unable to outperform TF-IDF.

ACKNOWLEDGEMENTS

This work has been sponsored by the RAAK-PRO program under grants of the EQuA-project. We would like to thank Alberto Bacchelli for his valuable suggestions.

REFERENCES

1. Lehman MM. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1984; **1**:213–221, doi:10.1016/0164-1212(79)90022-0.
2. Zaidman A, Pinzger M, van Deursen A. Software evolution. *Encyclopedia of Software Engineering*, Laplante PA (ed.). Taylor & Francis, 2010; 1127–1137, doi:10.1081/E-ESE-120044353.
3. Ernst NA, Borgida A, Jureta IJ, Mylopoulos J. Agile requirements engineering via paraconsistent reasoning. *Information Systems* 2013; (0):–, doi:10.1016/j.is.2013.05.008.
4. Scacchi W. Understanding the requirements for developing open source software systems. *IEE Proceedings - Software*, 2001; 24–39, doi:10.1049/ip-sen:20020202.
5. Ernst N, Murphy G. Case studies in just-in-time requirements analysis. *Int'l Workshop on Empirical Requirements Engineering (EmpiRE)*, IEEE, 2012; 25–32, doi:10.1109/EmpiRE.2012.6347678.
6. Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.* Jul 2002; **11**(3):309–346, doi:10.1145/567793.567795.
7. Heck P, Zaidman A. An analysis of requirements evolution in open source projects: Recommendations for issue trackers. *Int'l Workshop on the Principles of Software Evolution (IWPSE)*, ACM, 2013; 43–52, doi:10.1145/2501543.2501550.
8. Martakis A, Daneva M. Handling requirements dependencies in agile projects: A focus group with agile software development practitioners. *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, 2013; 1–11, doi:10.1109/RCIS.2013.6577679.
9. Gotel O, Cleland-Huang J, Hayes JH, Zisman A, Egyed A, Grünbacher P, Dekhtyar A, Antoniol G, Maletic JI, Mäder P. Traceability fundamentals. Cleland-Huang *et al.* [19]; 3–22.
10. Cavalcanti YC, da Mota Silveira Neto PA, Machado IdC, Vale TF, de Almeida ES, de Lemos Meira SR. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process* 2013; :n/a–n/adoi:10.1002/smr.1639. URL `http://dx.doi.org/10.1002/smr.1639`.
11. Cleland-Huang J, Dumitru H, Duan C, Castro-Herrera C. Automated support for managing feature requests in open forums. *Commun. ACM* 2009; **52**(10):68–74, doi:10.1145/1562764.1562784.
12. Gu H, Zhao L, Shu C. Analysis of duplicate issue reports for issue tracking system. *Int'l Conf on Data Mining and Intelligent Information Technology Applications (ICMiA)*, 2011; 86–91.
13. Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. *Proc. Int'l Conf. on Software Engineering (ICSE)*, IEEE, 2007; 499–510, doi:10.1109/ICSE.2007.32.
14. Sun C, Lo D, Wang X, Jiang J, Khoo SC. A discriminative model approach for accurate duplicate bug report retrieval. *Proc. Int'l Conf. on Software Engineering (ICSE)*, 2010; 45–54, doi:10.1145/1806799.1806811.
15. Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc YG. Is it a bug or an enhancement?: A text-based approach to classify change requests. *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, ACM: New York, NY, USA, 2008; 23:304–23:318, doi:10.1145/1463788.1463819. URL `http://doi.acm.org/10.1145/1463788.1463819`.
16. Ko AJ, Myers BA, Chau DH. A linguistic analysis of how people describe software problems. *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC)*, IEEE Computer Society, 2006; 127–134, doi:10.1109/VLHCC.2006.3. URL `http://dx.doi.org/10.1109/VLHCC.2006.3`.
17. Moreno L, Bandara W, Haiduc S, Marcus A. On the relationship between the vocabulary of bug reports and source code. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE, 2013; 452–455.
18. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 1990; **41**(6):391–407, doi:10.1002/(SICI)1097-4571(199009)41:6⟨391::AID-ASI1⟩3.0.CO;2-9.
19. Cleland-Huang J, Gotel O, Zisman A ((eds.)). *Software and Systems Traceability*. Springer, 2012.
20. Gotel OCZ, Finkelstein A. An analysis of the requirements traceability problem. *Proceedings of the First IEEE International Conference on Requirements Engineering (ICRE)*, IEEE, 1994; 94–101.
21. De Lucia A, Fasano F, Oliveto R. Traceability management for impact analysis. *Frontiers of Software Maintenance (FoSM)*, IEEE, 2008; 21–30, doi:10.1109/FOSM.2008.4659245.

22. Cleland-Huang J, Czauderna A, Gibiec M, Emenecker J. A machine learning approach for tracing regulatory codes to product specific requirements. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, ACM, 2010; 155–164.
23. Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on* 2002; **28**(10):970–983.
24. Marcus A, Maletic J. Recovering documentation-to-source-code traceability links using latent semantic indexing. *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003; 125–135, doi:10.1109/ICSE.2003.1201194.
25. De Lucia A, Fasano F, Oliveto R, Tortora G. Can information retrieval techniques effectively support traceability link recovery? *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006; 307–316, doi:10.1109/ICPC.2006.15.
26. Maarek YS, Berry DM, Kaiser GE. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Software Eng.* 1991; **17**(8):800–813.
27. Poshyvanyk D, Guéhéneuc YG, Marcus A, Antoniol G, Rajlich V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.* 2007; **33**(6):420–432.
28. Antoniol G, Canfora G, Casazza G, Lucia AD. Identifying the starting impact set of a maintenance request: A case study. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, 2000; 227–230.
29. Marcus A, Maletic JI. Identification of high-level concept clones in source code. *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2001; 107–114.
30. Maletic JI, Marcus A. Supporting program comprehension using semantic and structural information. *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2001; 103–112.
31. Qusef A, Bavota G, Oliveto R, Lucia AD, Binkley D. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software* 2014; **88**:147–168.
32. Lucia AD, Fasano F, Oliveto R, Tortora G. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.* 2007; **16**(4).
33. Marcus A, Maletic JI, Sergeyev A. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering* 2005; **15**(5):811–836.
34. Lormans M, van Deursen A, Groß HG. An industrial case study in reconstructing requirements views. *Empirical Software Engineering* 2008; **13**(6):727–760.
35. Binkley D, Lawrie D. *Maintenance and Evolution: Information Retrieval Applications*, chap. 49. 2011; 454–463, doi:10.1081/E-ESE-120044704. URL `http://www.tandfonline.com/doi/abs/10.1081/E-ESE-120044704`.
36. Hayes JH, Dekhtyar A, Osborne J. Improving requirements tracing via information retrieval. *2003 11th IEEE International Requirements Engineering Conference (RE)* 2003; **0**:138, doi:http://doi.ieeecomputersociety.org/10.1109/ICRE.2003.1232745.
37. och Dag JN, Gervasi V, Brinkkemper S, Regnell B. Speeding up requirements management in a product software company: Linking customer wishes to product requirements through linguistic engineering. *2012 20th IEEE International Requirements Engineering Conference (RE)* 2004; **0**:283–294, doi:http://doi.ieeecomputersociety.org/10.1109/ICRE.2004.1335685.
38. Cleland-Huang J. Traceability in agile projects. Cleland-Huang *et al.* [19]; 265–275.
39. Kulshreshtha V, Boardman JT, Verma D. The emergence of requirements networks: the case for requirements inter-dependencies. *IJCAT* 2012; **45**(1):28–41.
40. Sandusky RJ, Gasser L, Ripoche G. Bug report networks: Varieties, strategies, and impacts in a f/oss development community. *Proc. Int'l Workshop on Mining Software Repositories (MSR)*, 2004; 80–84.
41. Herzig K, Just S, Rau A, Zeller A. Classifying code changes and predicting defects using changegenealogies. *Technical Report*, Software Engineering Chair, Saarland University, Dept. of Informatics 2013. `https://www.st.cs.uni-saarland.de/publications/details/herzig-genealogytechreport-2011/`.
42. Herzig K, Just S, Zeller A. It's not a bug, it's a feature: How misclassification impacts bug prediction. *Technical Report*, Universitaet des Saarlandes, Saarbruecken, Germany August 2012.
43. Jalbert N, Weimer W. Automated duplicate detection for bug tracking systems. *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, 2008; 52–61, doi:10.1109/DSN.2008.4630070.
44. Tian Y, Sun C, Lo D. Improved duplicate bug report identification. *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2012; 385–390, doi:10.1109/CSMR.2012.48.
45. Sun C, Lo D, Khoo SC, Jiang J. Towards more accurate retrieval of duplicate bug reports. *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, IEEE, 2011; 253–262, doi:10.1109/ASE.2011.6100061.

46. Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. *Proc. Int'l Conf. on Software Engineering (ICSE)*, ACM, 2008; 461–470.
47. Dumais ST. Latent semantic analysis. *Annual Review of Information Science and Technology* 2004; **38**(1):188–230, doi:10.1002/aris.1440380105. URL `http://dx.doi.org/10.1002/aris.1440380105`.
48. Santhiappan S, Gopalan VP. Finding optimal rank for lsi models. *Proceedings of ICAET*.
49. Porter MF. An algorithm for suffix stripping. *Program* July 1980; **14**(3):130–137.
50. De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Applying a smoothing filter to improve ir-based traceability recovery processes: An empirical investigation. *Inf. Softw. Technol.* Apr 2013; **55**(4):741–754, doi: 10.1016/j.infsof.2012.08.002. URL `http://dx.doi.org/10.1016/j.infsof.2012.08.002`.
51. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering* 2009; **14**(2):131–164, doi:10.1007/s10664-008-9102-8.
52. Yin RK. *Case Study Research: Design and Methods, 5th edition*. Sage Publications, 2013.
53. Oliveto R, Gethers M, Poshyvanyk D, Lucia AD. On the equivalence of information retrieval methods for automated traceability link recovery. *Proceedings of the IEEE International Conference on Program Comprehension (ICPC)*, IEEE Computer Society, 2010; 68–71.