Traceability in the Wild: Automatically Augmenting Incomplete Trace Links

Michael Rath[1], Jacob Rendall[2], Jin L.C. Guo[3], Jane Cleland-Huang[2], Patrick Mäder[1]
Technical University Ilmenau, Ilmenau, Germany[1]
University of Notre Dame, South Bend, USA[2]
McGill University, Montreal, Canada[3]
{michael.rath,patrick.maeder}@tu-ilmenau.de;{jrendal1,JaneClelandHuang}@nd.edu;jguo@cs.mcgill.ca

ABSTRACT

Software and systems traceability is widely accepted as an essential element for supporting many software development tasks. Today's version control systems provide inbuilt features that allow developers to tag each commit with one or more issue ID, thereby providing the building blocks from which project-wide traceability can be established between feature requests, bug fixes, commits, source code, and specific developers. However, our analysis of six open source projects showed that on average only 60% of the commits were linked to specific issues. Without these fundamental links the entire set of project-wide links will be incomplete, and therefore not trustworthy. In this paper we address the fundamental problem of missing links between commits and issues. Our approach leverages a combination of process and text-related features characterizing issues and code changes to train a classifier to identify missing issue tags in commit messages, thereby generating the missing links. We conducted a series of experiments to evaluate our approach against six open source projects and showed that it was able to effectively recommend links for tagging issues at an average of 96% recall and 33% precision. In a related task for augmenting a set of existing trace links, the classifier returned precision at levels greater than 89% in all projects and recall of 50%.

KEYWORDS

Traceability, Link Recovery, Machine Learning, Open Source

ACM Reference Format:
Michael Rath, Jacob Rendall, Jin.L.C.Guo, Jane Cleland-Huang, Patrick Mäeder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of ICSE'18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE'18),* 12 pages.
https://doi.org/10.1145/3180155.3180207

1 INTRODUCTION

Traceability provides support for many different software engineering activities including safety analysis, change impact analysis, test regression selection, and coverage analysis [8, 21, 22, 39, 40, 48]. Its importance has long been recognized in safety-critical domains, where it is often a prescribed part of the development process [9, 42, 51, 52, 55]. While traceability is relevant to all software development environments [41–43, 48, 53, 61], the effort needed to manually establish and maintain trace links in non-regulated domains has often been perceived as prohibitively high.

However, with the ubiquitous adoption of version control systems such as *Git* [20] and *GitHub*, and issue tracking systems such as *Bugzilla* or *Jira* [30], it has become common practice for developers to tag commits with issue IDs. In large projects, such as the ones from the Apache Foundation, this procedure is reflected in the guidelines which state that *"You need to make sure that the commit message contains at least [...] a reference to the Bugzilla or JIRA issue [...]"* [18]. Creating such tags establishes explicit links between commits and issues, such as feature requests and bug reports. However, the process is not perfect, as developers may forget, or otherwise fail, to create tags when they make a commit [4, 56]. While the practice of tagging commits has become popular in open source projects, it is conceptually applicable in any project where version control systems and issue trackers are used.

In this paper we propose a solution for identifying tags that are missing between commits and issues and augmenting the traceability data with these previously missing links. As shown later in the paper, our observations across six OSS showed that an average of only about 60% of commits were linked to specific issues. The majority of papers addressing traceability in OSS have focused on directly establishing a complete set of links between issues and source code. In contrast, we focus on generating the missing links at the commit level. This has the primary advantage of providing traceability support within the natural context in which developers are creating trace links. Our approach leverages existing tags, as well as information related to the commit process itself and also textual similarities between commit messages, issue descriptions, and code changes. We use these attributes to train a classifier to identify tags that are missing from commit messages. Furthermore, we set a critical constraint on our work that the classifier must be populated, trained, and then utilized with a simple "button press" in order to make it practical in an industrial setting.

Low level links between commits and issues provide the building blocks for inferring project-wide traceability between improvements, bug reports, source code, test cases, and commits, and also allow associations to be established between the issues and developers [58]. Augmenting the set of trace links between commits and issues, therefore results in a more complete set of project-wide trace links. This enables more accurate support for tasks such as

ICSE'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180207

Issue ID:	GROOVY-5223	**Type:**	Improvement
Summary:	[GROOVY-5223] Bytecode optimizations: make use of LDC for class literals		
Description:	Class literals are currently loaded using generated getclass$ methods which increase bytecode size and may prevent some optimizations. In most situations though, we may use the LCD bytecode instruction to load the class literal.		
Status:	Closed	**Created:**	30/Dec/11 09:59
Resolution:	Fixed	**Resolved:**	03/Jan/12 02:29

Figure 1: Example of an improvement in Jira issue tracker

defect prevention [53], change impact analysis, coverage analysis, and even provides enhanced support for building recommendation systems to identify appropriate developers for fixing bugs [2].

We train and evaluate our approach on six open-source projects in order to address three key research questions:

RQ1: Is the link classifier able to accurately reconstruct issue tags during the commit process?

RQ2: Is the link classifier able to precisely augment an existing set of incomplete commit to issue links in a fully automated way?

RQ3: Is the link classifier able to recommend additional tags?

The remainder of the paper is structured as follows. Section 2 introduces the artifacts, case projects, process model, and stakeholder model, that form the fundamentals of our approach. Section 3 describes the elements of our classifier. Section 4 describes the six projects in our study. Sections 5 and 6 describe scenarios and experiments associated with recommending tags for commits, augmenting existing sets of trace links, and constructing trace links for commits with no tags. Finally sections 7 to 9 discuss related work, threats to validity, and conclusions.

2 FUNDAMENTALS

We first introduce a motivating example and describe the artifacts, project environments, and the process and stakeholder models that form the fundamentals of our approach.

2.1 Motivating Example

Figure 1 depicts the improvement request, GROOVY-5223,[1] retrieved from the Groovy project's issue tracker, JIRA [30]. The request consists of a unique issue ID, a short summary, a longer textual description, time stamps for issue creation and resolution, the issue's current status, and information about its resolution. This particular improvement requests an enhancement to an existing feature concerning class loading at byte code level. Figure 2 shows a bug report GROOVY-5082[2] for the same project. It includes the same fields as the improvement, except that the type is specified as a bug. In this case, the bug describes a problem with byte code generation for the groovy language. Finally, Figure 3 shows an example of a commit[3] submitted to the Git [20] version control system. A commit (change set) includes a unique commit hash value, a message describing its purpose, the time stamp when it was submitted, and finally a list of files modified by the change set.

[1] https://issues.apache.org/jira/browse/GROOVY-5223
[2] https://issues.apache.org/jira/browse/GROOVY-5082
[3] http://goo.gl/pBy6Nw

Issue ID:	GROOVY-5082	**Type:**	Bug
Summary:	[GROOVY-5082] Sometimes invalid inner class reference left in .class files produced for interfaces		
Description:	Compile this: *[...]* Upon javap'ing the result we see this InnerClass attribute: ... to X$1. But there is no X$1 produced on disk....		
Status:	Closed	**Created:**	17/Oct/11 13:45
Resolution:	Fixed	**Resolved:**	01/Feb/12 11:10

Figure 2: Example of a resolved bug in Jira issue tracker

Hash:	b1bb2abfde414950238ff4d895bf5e182793500a
Message:	GROOVY-5082: remove synthetic interface loading helper class in case it is not used
Committed:	Feb 1, 2012
Files:	src/main/org/codehaus/groovy/classgen/AsmClassGenerator.java

Figure 3: Example for a commit in Git

The common way to establish a trace link between a commit and an issue is by placing the unique issue id, i.e., GROOVY-5082 in this example, into the beginning of the commit message. However, a close examination of the commit message in this example shows that the committer made a subtle mistake and misspelled the issue key for the bug that was being fixed (omitting an **O**). As a result, traditional trace link construction techniques that rely upon matching the key to an issue will fail to create a trace link.

However, even without a valid issue key, there are numerous clues to suggest that the commit should be associated with the reported bug. First, the bug description exhibits textual similarity to the commit message as well as to the text in the changed file AsmClassGenerator.java. Second, the commit was submitted on the same date that the issue was resolved, and finally, the person (obfuscated for privacy reasons) who submitted the commit was also responsible (i. e. the assignee) for resolving the issue. Taken together, these observations provide some degree of evidence that the commit and bug should be linked. This example illuminates the thinking behind our proposed solution. We build a classifier that leverages all of this information, plus additional attributes, to learn which issues should be tagged to each specific commit.

2.2 Software Artifacts and their Relations

While version control systems and issue trackers have several types of artifacts, our approach leverages three of them to construct missing commit links. These are *issues*, *commits* (i.e., change sets), and *source code files*.

Issues: Our model uses issues collected from the Jira issue tracking system. While there are several types of *issues*, we focus on *improvements* and *bugs* which are the most commonly occurring ones. An improvement represents an enhancement to an existing feature in the software, while a bug describes a problem, which impairs or prevents its correct functionality. In the remainder of the paper, the term *issue* is used in reference to both improvements and bugs. Independent of their actual type, all issues share the following properties: a unique *issue ID*, a *summary* providing a brief one-line synopsis of the issue, and a more extensive explanation provided in the *description*. Further, every issue has a temporal life cycle – it is *created* at a given point in time and later *resolved*, and may be assigned to an author, responsible for its resolution.

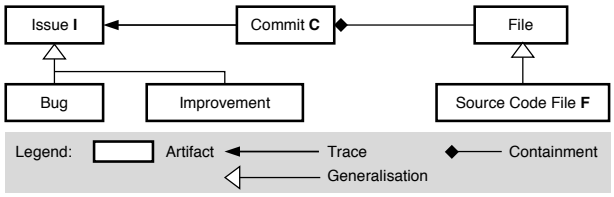

Figure 4: Studied artifact model with issues $\mathcal{I}$, commits C, source code files $\mathcal{F}$, and their relations.

Commit (Change Set): In Git version control, changes are organized as atomic *commits*. A commit bundles together all modified files and is uniquely identified by a *hash* value. It includes properties concerning the person who made the change, a time stamp (committed date), and a commit message stating the purpose of the change.

Source Code: Multiple types of files are associated with a software project including *source code files*, *documentation*, *examples*, and *tests*. In this paper we focus on source code files which are explicitly linked to commit messages. The source code files provide support for our primary goal of establishing links between commit messages and issues.

Relations: We are also interested in relations between the three fundamental types of artifacts in our model. A commit atomically bundles one or more source files together. This containment relation between commit and source code artifacts is a natural result of submitting the change set to the version control system. Further, as previously explained, trace links are explicitly created between issues and commits when a developer tags a commit with a valid issue ID. We denote an issue as *linked*, if there is at least one trace link from the issue to a commit. Issues without any links are termed *non linked*. Figure 4 depicts the three artifacts as well as their structural interactions. We denote $\mathcal{I} = \mathcal{I}_{Bug} \cup \mathcal{I}_{Imp}$ as the set of issues (bugs and improvements), C the set of commits, and $\mathcal{F}$ the set of source code files in a project. The function $is\_linked : C \times \mathcal{I} \rightarrow \{0, 1\}$ returns 1 if an explicit link exists, and 0 otherwise. The function $mod(C) : C \rightarrow F_C$ with $F_C \subseteq \mathcal{F}, C \in C$, calculates this set for a given commit. A source code file may be part of multiple commits.

2.3 Studied Projects

For our study, we selected six projects from diverse domains, that utilized both Git and Jira. They included: build automation (Maven (Ma)), databases (Derby (De), Infinispan (In)), languages (Groovy (Gr), Pig (Pi)), and a rule engine (Drools (Do)), primarily selected because each of these projects has existed for several years, has a non-trivial number of commits and issues, and largely followed the practice of tagging commits with issue IDs. We analyzed each of the projects to gain an understanding of the numbers of links that existed between commit messages and issues. Further, we analyzed the number of issues that were linked to exactly one commit (1:1), two or more disjoint commits (1:N), or had no links. Results are reported in Table 1. For example, of the 2,638 bug-related issues in the Derby project, 1,093 were linked to only one commit, 273 were linked to multiple commits, and 1,272 had no associated commits.

Table 1: Existing bug and improvement to commit link characteristics in the studied project.

		Project					
Link Type	Profile	De	Dr	Gr	In	Ma	Pi
1:1	Bug	1093	1040	1609	1714	609	1066
	Imp.	399	61	459	393	222	313
1:n	Bug	273	236	422	254	99	87
	Imp.	225	28	179	96	46	48
Non	Bug	1272	605	1667	994	769	944
	Imp.	730	42	392	157	313	251

Table 2: Existing commit to bug and improvement link characteristics in the studied project.

		Project					
Link Type	Profile	De	Dr	Gr	In	Ma	Pi
1:1	Bug	1657	1559	2423	2223	781	1206
	Imp.	1350	195	903	671	335	437
1:n	both	175	138	139	95	48	38
Non		553	2947	4740	1479	3614	64

Across all of the projects approximately 43.3% of improvements and 42.4% of bugs have no commits associated with them.

Table 2 depicts a similar analysis from the perspective of the commits. It reports the number of commits with links to issues for the selected projects. Again, we analyzed the distribution of 1:1 links, 1:N links and non linked commits. In the Derby project, of the 3,735 commits, 1,657 linked to only one bug, 1,350 to only one improvement, 175 linked to multiple bugs or improvements, and 553 commits had no links. However, across all of the projects approximately 48% of the commits were not linked to any issue. Furthermore, there was significant variance across the six projects with only 15% of commits in Derby having no links compared to approximately 76% of unlinked commits in Maven. Clearly, different practices exist across different projects, leading to huge disparities in the extent to which issue tags are added to commit messages.

One of the primary goals of our work, is to establish at least one link for each commit. As the majority of commits link to a single issue, we only attempt to generate links for currently unlinked commits. For commits without links there are two viable cases – first that an appropriate issue exists and a link can be generated, and second that no appropriate issue exists for the commit.

2.4 Process Model

As previously explained, our approach leverages clues from the development process to aid in the generation of links. First we observe that the software development process is time dependent: bugs and improvements are constantly created and resolved, and commits are submitted to the version control system. Figure 5 exemplifies this scenario. It contains six issues $\mathcal{I} = \{I_1 \ldots I_4, B_1, B_2\}$, nine commits $C = \{C_1 \ldots C_9\}$, and six source code file artifacts $\mathcal{F} = \{F_1 \ldots F_6\}$. The issue artifacts and commits are ordered across a time line. In this example, the issues I_1, I_2, I_4, B_1, as well as commits C_1, C_5, C_8, and

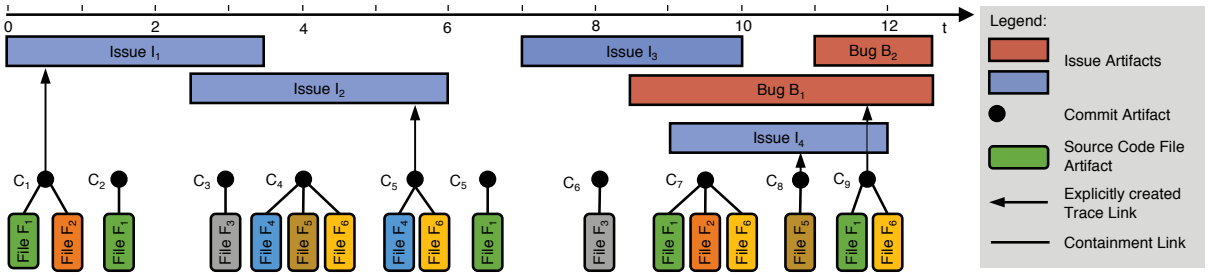

Figure 5: Temporal and structural relations between issue ($\mathcal{I}$), commit (C), and source code file ($\mathcal{F}$) artifacts.

C_9 are linked, e. g. $is\_linked(C_1, I_1) = 1$. The figure also shows the relation between issues and commits according to the time line. We define the functions $created : I \rightarrow \mathbb{N}$ and $resolved : I \rightarrow \mathbb{N}, I \in \mathcal{I}$, which returns the point in time when the issue was created and respectively resolved. During this time, the issue is considered to be unfinished and source code modifications are required in order to implement the improvement or fix the bug. In our study we focus on issues that are resolved (e. g., in Figure 5, $created(I_1) = 0$ and $resolved(I_3) = 10$). The function $committed : C \rightarrow \mathbb{N}, C \in \mathcal{C}$ returns the time stamp at which a commit was submitted to the version control system. e. g. $committed(C_6) = 8$ in the example.

Temporal relations Considering a non linked commit $C \in \mathcal{C}$, the temporal structure imposes several constraints on the possible link candidates $I \in \mathcal{I}$. The following three cases exist.

(1) $committed(C) < created(I)$: Due to causality, the commit C is not considered to be a link candidate for I (e.g., in Figure 5, C_2 is not a link candidate for I_2).
(2) $created(I) \leq committed(C) \leq resolved(I)$: This situation depicts the usual development work flow. After issue creation, the developers modify the source code and submit commits in order to resolve the issue. These commits are traced to the issue. Eventually, the issue is resolved, and in this example, no further commits are made to the issue (e.g., in Figure 5, the non linked commit C_6 is a link candidate for I_3).
(3) $resolved(I) < committed(C)$: Intuitively, in this situation a trace link from C to I is not considered, since I was already resolved before the commit occurred. However, this situation is not uncommon as Table 3 shows. The obvious reasons might be, that a developer forgot to submit the commit before resolving the issue. Another might simply be clock differences between the unconnected, decentralized systems used by Jira and Git which prevents strict time comparisons.
In project Derby, there is sometimes a large discrepancy between the time at which an issue is resolved and the last commit that traces to it. For example, the improvement `DERBY-6516`[4] was resolved as fixed on 20/Mar/14; yet, on 4/Apr/14 a commit (`78227e4`[5]) was submitted and linked to this improvement. However this scenario is quite rare, affecting only 136 commits. Interestingly, in both the Groovy and Maven projects, the median time difference for late commits is much lower (only 5 hours), but affects a huge number of commits. For example, in the Maven project, we observed that between 2005 and 2015 there was a constant offset between issue resolution and corresponding commit from either five or six hours as illustrated in `MNG-221`[6] (from 2005), `MNG-2376` (from 2008), and `MNG-5245` (from 2012).

[4]https://issues.apache.org/jira/browse/DERBY-6516
[5]http://goo.gl/j3WYd6

Table 3: Properties of linked commits. (1) The distribution of commits linked to issues after issues resolution along with the median time. (2) The average file overlap of consecutive commits linked to the same issue.

	Project					
	De	Dr	Gr	In	Ma	Pi
(1) Commits linked to already resolved issues						
Number	136	207	2,648	244	847	100
Median time after resolved	150h	60h	5h	19h	5h	60h
(2) Avg. file commit overlap	0.35	0.35	0.71	0.33	0.40	0.45

These temporal constraints limit the potential pairs of candidate links between non linked commits and non linked improvements and bugs.

Structural relations Table 2 reveals (in row 1:n) that often multiple commits $C_a, C_b, \ldots$ are required in order to solve an issue I. Ideally all of these commits are traced to the respective issue. However, often only one commit in this series is explicitly linked to I. In [57] the other commits in this series are termed *phantoms*. All commits in the series may share commonalities. In addition to their succession in time, the commits may modify a similar set of source code files since they are related to the same issue. We define a function $overlap(C_a, C_b) = \frac{mod(C_a) \cap mod(C_b)}{\max(|mod(C_a)|, |mod(C_b)|)}$ with $C_a, C_b \in \mathcal{C}$. For example, in Figure 5 the overlap of $C1$ and $C2$ is $overlap(C_1, C_2) = \frac{1}{2}$, and $overlap(C_3, C4) = 0$. As shown in Table 3, the average overlap of consecutive commits linked to the same issue varies among the projects. For example in Derby, the average overlap is 0.35 meaning, that, on average, one out of three files are the same for commits in a series. The highest number, 0.73, is achieved in Groovy, where a few files are changed multiple times to implement an improvement or bug. Three commits (`51d4fee`[7], `3d20737`[8], and `974c945`[9]) were submitted between 4/Jan/2009 and 6/Jan/2009 all modifying one and the same source

[6]https://issues.apache.org/jira/browse/MNG-221
[7]http://goo.gl/4YGjhe
[8]http://goo.gl/AZcwmK
[9]http://goo.gl/2TYhsz

code file `DefaultGroovyMethods.java` and linked to improvement `GROOVY-3252`[10]. This results in $overlap = 1$ for each commit pair in the series.

Based on temporal closeness and overlap, there are indications that C_2 and C_1 may belong to a series of commits and thus could be traced to I_1. The situation may also occur forward in time, i. e. C_7 and C_9 may belong to a series because of temporal closeness and source file overlap and thus should be traced to bug B_1.

2.5 Stakeholder Model

Issues and commit artifacts both carry information about the author. The assignee of an issue also might be the person who contributes commits in solving the issue. In the studied scenario, there is no technical connection between the issue tracker Jira and the version control system Git. Thus we cannot rely on an available stakeholder model. Therefore we applied the following approach to identify individual developers in both systems. In each system, a developer is represented by a name and a login (nickname or email). In the first step, we separately collected all developers from the two systems and built two groups. In this step, we merged names if they used the same login and therefore were aliases for the same person. In the second step, we heuristically merged the two resulting developer lists and compared the names, in order to identify the same person in both systems. In order to fully protect user privacy and to comply with Github Privacy Requirements a unique number, *user id*, was assigned to every developer. The function $userid : u \rightarrow \mathbb{N}$ with $u \in \mathcal{C} \cup \mathcal{I}$ returns this user id for a given commit or issue.

3 THE LINK CLASSIFIER

Our goal was to create a classifier that could identify issues associated with a commit. The classifier was therefore trained to predict whether any issue-commit pair should be linked or not.

3.1 Attributes of the Commit-Issue Relation

Based on the artifact model introduced in the previous section, we identified 18 attributes per instance. These attributes fall into two categories, process-related information and textual similarity between artifacts using information retrieval techniques.

Process-Related Attributes

We consider the following 16 process-related factors to model the relationship between commits, source code files, and issues. These factors capture stakeholder-related, temporal, and structural characteristics of the candidate pair (C, I) with $C \in \mathcal{C}, I \in \mathcal{I}$:

Stakeholder-related information, $a_{1\ldots3}$**:** We capture the identities of the committer as $a_1 = userid(C)$ and the assignee of the issues as $a_2 = userid(I)$. Additionally, we marked as a binary attribute whether the two are identical as $a_3 = 1$ (if $userid(C) = userid(I)$, 0 otherwise).

Temporal relations between issue and commit, $a_{4\ldots7}$**:** Based on temporal properties of issue and commit, we calculated $a_4 = committed(C) - created(I)$ and $a_5 = resolved(I) - committed(C)$. Additionally, we capture as a_6 whether $created(I) \leq committed(C) \leq resolved(I)$, i. e. whether C was committed during the active development time of I. Furthermore, we capture close commits in relation to issue resolution as $a_7 = |a_5| < \epsilon$. We set $\epsilon = 2.5$ days, derived from observing that late commits occur on average within 5 and 150 hours of the issue resolution for the studied projects (see Table 3). For example in Figure 5, the pair (C_6, I_3) yields $a_4 = 1$, $a_5 = 2$ and $a_6 = 1$.

Closest previous linked commit, $a_{8\ldots10}$**:** We capture the set of previous commits linked to I as $C_{prev} = \{C_x | is\_linked(C_x, I) \wedge committed(C_x) < committed(C)\}$. If non-empty, the commit $C_p \in C_{prev}$ with the largest commit time stamp is taken and used to calculate $a_8 = committed(C) - committed(C_p)$, $a_9 = overlap(C_p, C)$, and $a_{10} = userid(C_p)$. For example in Figure 5, the pair (C_2, I_1) yields $C_{prev} = \{C_1\}$ and thus $C_p = C_1$, $a_8 \approx 1$, and $a_9 = \frac{1}{2}$.

Closest subsequent linked commit, $a_{11\ldots13}$**:** Analogous to the closest previous linked commit, we capture subsequent commits C_{next}. We capture $C_{next} = \{C_x | is\_linked(C_x, I) \wedge committed(C) < committed(C_x)\}$ and selected C_n with the minimal commit time to calculate $a_{11} = committed(C_n) - committed(C)$, $a_{12} = overlap(C_n, C)$, and $a_{13} = userid(C_n)$. For example in Figure 5, the pair (C_7, B_1) yields $C_{next} = \{C_9\}$, $C_n = C_9$, $a_{11} \approx 2$, and $a_{12} = \frac{2}{3}$.

Number of issues and existing links, $a_{14\ldots16}$**:** We calculate the set of existing issues at time $committed(C)$, $I_{exist} = \{I_x | created(I_x) \leq committed(C) \leq resolved(I_x) \wedge I_x \in \mathcal{I}\}$ and capture its cardinality as $a_{14} = |I_{exist}|$. Taking I_{exist}, we derive $I_{user} = \{I_x | I_x \in I_{exist} \wedge userid(I_x) = userid(I)\}$ representing non-resolved issues for the assignee of I at that instant in time and capture its size in $a_{15} = |I_{user}|$. With $a_{16} = |\{C_x | is\_linked(I, C_x) \wedge committed(C_x) < committed(C), \forall C_x \in \mathcal{C}\}|$ we capture the number of links to I before commit C. For example in Figure 5, considering pair (C_7, B_1), $I_{exist} = \{B_1, I_3, I_4\}$ and thus $a_{14} = 3$.

Textual Similarity Attributes

We leveraged information retrieval methods to compute textual and semantic associations between commit messages, source code files, and issues. We explored three primary techniques for computing textual similarity *sim*. These were the Vector Space Model (VSM), VSM with N-Gram enhancements (VSM-nGram), and Latent Semantic Indexing (LSI) [1, 12, 14].

In the VSM model, each document, i.e., commit message, issue description, and source code file, is treated as an unstructured bag of terms. Following common information retrieval techniques, documents are pre-processed to remove stop words, to stem words to their morphological roots, and to split camel-case and snake-case words (e.g., optionsParser vs. options_parser) into their constituent parts. Each document d is then represented as a vector $\vec{d} = (w_{1,d}, w_{2,d}, ..., w_{n,d})$, where $w_{i,d}$ represents the term weight associated with term i for document d. Each term t is assigned a weight using a standard weighting scheme known as $tf - idf$ [29]. The cosine similarity between a pair of vectors in then computed as follows in order to estimate the similarity between two documents $d1$ and and $d2$:

$$sim(d1, d2) = \frac{(\sum_{i=1}^{n} w_{i,d1} w_{i,d2})}{\left(\sqrt{\sum_{i=1}^{n} w_{i,d1}} \cdot \sqrt{\sum_{i=1}^{n} w_{i,d2}}\right)} \quad (1)$$

[10] https://issues.apache.org/jira/browse/GROOVY-3252

The N-Gram enhancement to VSM utilizes n-gram models [7, 63]. N-gram is a contiguous sequence of n words in a document. Each document is again represented as a vector, but in this case, the vector is comprised of both the word and the n-grams it contains. The documents are preprocessed in the same way as the basic VSM. Based on initial experimentation, we set n from 2 to 4, to include 2-gram, 3-gram and 4-gram sequences in the vector representations. The similarity between vectors was again calculated using the cosine measure (equation (1)) with $tf - idf$ schema as described above.

We conducted an initial comparative study of Latent Semantic Indexing (LSI) [1, 12, 14], VSM, and VSM-nGram. Based on an initial comparison of the results we selected the VSM-nGram approach for computing textual similarity scores. This was because we observed that VSM-nGram outperformed VSM on our datasets, and ran much faster than LSI. In fact, the computation time of LSI on our datasets was prohibitively slow with runtimes of up to 40 hours in some cases, and so we rejected it as impractical. Furthermore, several previous studies have shown that VSM tends to either outperform LSI on software engineering datasets or perform in equivalent ways [25, 29, 38]. A detailed comparison of trace retrieval techniques within our classifier is outside the scope of this research. Therefore, based on our initial analysis, we chose VSM-nGram to compute the following similarity attributes:

Textual similarity of a commit and an issue, a_{17}: The similarity between the commit message and the textual content of the issue (for both improvements and bugs) is captured as $a_{17} = sim(C, I)$ with $C \in \mathcal{C}, I \in \mathcal{I}$.

Textual similarity of committed source files and an issue, a_{18}: For each commit-issue pair, the textual similarity between the content of the most similar committed source code file and the textual content of the issue is captured as $a_{18} = \max\{sim(F, I) | \forall F \in mod(C)\}, C \in \mathcal{C}, I \in \mathcal{I}$.

3.2 Studied Attribute Sets

We studied the impact of the presented attributes in four subsets.

- **Process** – This set solely contains the process-related attributes, i. e. $A_{Structure} = \{a_1 \dots a_{16}\}$. It studies the impact of all process-related attributes without considering textual similarity.
- **Similarity** – This set consists of the attributes $A_{Sim} = \{a_6, a_{17}, a_{18}\}$. It solely considers textual similarity between commit and issue given the constraint that the issue existed at the time of the commit.
- **All** – This set, $A_{all} = \{a_1 \dots a_{18}\}$, contains all process, similarity and stakeholder related attributes.
- **Auto** – This set, $A_{auto} \subseteq A_{all}$, addresses potential correlations and dependencies among attributes. It contains an automatically selected subset derived by considering the individual predictive ability of each attribute along with the degree of redundancy between them. We implemented the redundant attribute removal process based on Weka's inbuilt auto-selection feature [28].

3.3 Dataset Profiles and Splits

We aim to classify links between commits and improvements and between commits and bugs. We therefore construct two distinct profiles for each project, constructed from process and similarity attributes per commit-issue pair.

$$Profile_p = (S_{p,train},\ S_{p,test}) \qquad p \in \{Bug, Imp\}$$

Each profile consists of a distinct training and a testing set. We applied the following procedure per project to create instances of candidate commit–issue pairs for the training set $S_{p,train}$ as well as the testing set $S_{p,test}$.

$$S_{p,t} = \{(C, I) \mid is\_candidate(C, I)\} \qquad C \in \mathcal{C}_{p,t},\ I \in \mathcal{I}_{p,t}$$
$$p \in \{Bug, Imp\} \qquad t \in \{train, test\}$$

with the function $is\_candidate$ defined as

$$is\_candidate(C, I) = created(I) \leq committed(C) \wedge committed(C) \leq resolved(I) + \epsilon$$

The function limits the number of candidate commit/issue pairs according to causality. A link candidate is never considered between a commit if the issue has not been created at the time of the commit. Secondly, ϵ assures that a commit is not unboundedly considered as a candidate for issues resolved in the past. Based on an analysis of commits onto closed issues (see Table 3), we found that the median commit time after the issue has marked resolved was between 5 and 150 hours for the studied projects and we decided to choose ϵ as 30 hours. The candidate sets $\mathcal{C}_{p,t}$ and $\mathcal{I}_{p,t}$ were then created as

$$\mathcal{C}_{p,train} = \{C \mid committed(C) \leq t_{split}\} \qquad C \in \mathcal{C}$$
$$\mathcal{I}_{p,train} = \{I \mid resolved(I) \leq t_{split}\} \qquad I \in \mathcal{I}_p$$
$$\mathcal{C}_{p,test} = \{C \mid committed(C) > t_{split}\} \qquad p \in \{Bug, Imp\}$$
$$\mathcal{I}_{p,test} = \{I \mid created(I) > t_{split}\}.$$

The parameter t_{split} defines the point in time, which splits the training and test set. We choose a 80% – 20% split and calculated t_{split} as follows. First, we ordered the improvements in the respective project according to their creation date in ascending order. We selected the improvement $I_{split} \in \mathcal{I}_{Imp}$, which divides this sequence into 80% and 20% of all improvements where $t_{split} = resolved(I_{split})$, i. e. the resolution time of 80% of all improvements.

Each commit-issue candidate in the profiles $Profile_{Bug}$ and $Profile_{Imp}$ forms an instance to train ($S_{p,train}$) or test ($S_{p,test}$) the classifier, where the test data (i. e. 20% in every project) is distinct from the training data. For each instance, we calculate 18 attributes $a_{1\dots18}$ that characterize the relation between commit and issue. In addition, each instance is annotated with the known class (i.e. *linked*, or *non-linked*) as extracted from the projects' data. *Linked* means that the developer had created an explicit tag from the commit to the issue, while *non-linked* means that no such tag exists.

3.4 Classifier Training

We investigated three different supervised learning classifiers for categorizing commit-issue pairs as *linked* or *non-linked*. These were Näive Bayes, J48 Decision Tree, and Random Forrest. We included Näive Bayes because even though the assumption of independence rarely holds in software project data, the algorithm has been demonstrated to be effective for solving similar prediction problems [11, 16, 26, 32]. We utilized Weka's J48 decision tree with default pruning settings because of its previously reported effectiveness in other software engineering studies [26]. Finally, we

included the Random Forest classifier because it has been shown to be highly accurate and robust against noise [5], although it can be expensive to run on large data sets.

The profiles we created were severely unbalanced containing many more instances of *non-links* than *links*. Training against such unbalanced sets makes it likely that the classifier will favor placing instances into the majority class (i. e. in this case classifying all pairs as *non-links*). We performed all experiments using Weka [27] and used the inbuilt sub-sampling feature to create balanced data sets. Given a fixed number of explicit *links*, Weka randomly selects the same number of *non-links*. We trained each classifier in turn using the balanced sets $S_{Bug,train}$, and $S_{Imp,train}$ for each project and then evaluated the classifier against the respective unbalanced testing sets $S_{Bug,test}$ and $S_{Imp,test}$. To mitigate the random effects of sub-sampling, we repeated the training and testing 10 times and averaged the achieved results. We did not follow an ordinary 10-fold cross validation approach because several of the studied variables (e. g. attributes $a_4 \ldots a_7$) reflect temporal sequences in the development, making it necessary to ensure that temporal sequencing between training and test data was preserved. For each technique the classifier returned a category (i.e. linked or non-linked) and also a score which we used to rank recommended links in order of likelihood.

4 DATA COLLECTION

To prepare the data for training and testing the classifier, we performed a two step data collection process for each of the six projects.

Step 1: Analyzing project management and issue tracker system. We implemented a collector to retrieve artifacts (i. e., improvements and bugs). All six projects use the Jira project management tool offering a web-service interface. Our collector downloaded and parsed all artifacts. Using the artifact type, we filtered the artifacts to retrieve only bugs and improvements. Therefore we applied the following mapping from Jira types to our model: bug → bug, and improvement, enhancement → improvement. In both cases, the artifacts represented finished work, i. e. their status was "Resolved" or "Closed" and the resolution "Fixed" or "Done".

Step 2: Analyzing Source Control Management (SCM) system. A second collector was implemented to download all source code changes and commit messages from each SCM repository (i.e., GIT). We parsed the commit messages and applied the heuristic described in [4] to retrieve existing trace links from commits to bug reports and improvements based on searching and matching the issue keys in commit messages. Given the goals of our traceability experiment we excluded non-source code files related to documentation, and build automation based on their file name extensions. Additionally we analyzed file paths in order to exclude source code files implementing test cases. In the standard Maven directory layout[11], used by all six of our projects projects, source files are placed in sub-directories of `src/main` and tests as sub-directories of `src/test/java`.

The results of these two steps were stored in an archive per project, which is publicly available [50]. Data were collected from each project until May 31$^{\text{th}}$ 2017.

[11] https://goo.gl/D8uYaD

5 RECONSTRUCTING KNOWN LINKS

We performed a two-phase evaluation. In the first phase we address RQ1 and RQ2 by exploring two different usage scenarios. The first uses the classifier as a *recommender* system, to suggest a list of the most likely issues at the time a commit is submitted. Ideally this functionality would be integrated into the version control system and activated when the user presses the commit button. In this scenario, high recall is imperative, so that the relevant issue (if it exists) is included in the displayed list. The second experiment evaluates the case, in which the classifier is used to *automatically augment* an existing set of trace links for a project. In this scenario, high precision is essential because links that are automatically added must exhibit high accuracy. In experiments related to both of these scenarios, we leveraged the existing links created by the project developers from explicit commit-issue tags as an "answer set" to train and evaluate our classifiers. Both experiments therefore evaluate whether the classifier would have been able to recommend or create a known link if the committer had forgotten to create its tag manually. We trained the three classifiers on the four attribute sets as described in Section 3.2 and Section 3.4.

Results were evaluated using commonly adopted traceability metrics. *Recall* measures the fraction of relevant links that are retrieved while *precision* measures the fraction of retrieved links that are relevant. Finally, *F-measure* measures the harmonic mean of recall and precision [1, 29, 36, 59]. We utilize two variants of the F-Measure – namely F_2 which is weighted to favor recall, and $F_{0.5}$ which is weighted to favor precision.

Scenario 1: Recommending Issues to Assist Commits: The goal of this scenario is to create a short list with a maximum of three recommended links, to assist developers in tagging their new commits with issue IDs. Thus, we truncate the retrieved lists after the third rank and evaluate classifier performance in terms of precision, recall, and F2-measure at this point. F2-measure is selected because the objective of this scenario is to achieve high recall. The results for the best performing classifier Random Forest are shown in Table 4. The attribute set *All* achieves an average recall of 96% and average precision of 33%, which in combination marks the best performance of the studied feature sets. An application of the Mann-Whitney U Test [44] shows that the *All* approach significantly ($p < 0.05$) outperforms the other attribute sets in terms of F_2 score. The other two classifiers also performed best when using the *All* features sets. However, their achieved F_2 scores were significantly lower than that of Random Forest ($\overline{F_2} = 0.48$ for J48, and $\overline{F_2} = 0.6$ for Näive Bayes). Generally, the values show that the Random Forest classifier is able to predict the one true link among the three recommended links. The attribute set *Similarity* exhibits the lowest F_2 measure. The feature set *Process* performs considerably well. This is notable, because it does not require resource intensive IR techniques to extract the necessary features a_{17} and a_{18}. However, adding these features to the model results in overall better performance (see the *All* attribute set). An exception within the results is the Derby project, which underperforms on all attribute sets. The low recall values indicate (e g., 0.56 for *Process*) that the correct link is not in the ranked list for one out of two commits.

Scenario 2: Fully Automated Augmentation of Trace links between Commits and Issues: The classifier performance for

Table 4: Trace recommender evaluation using the different attribute sets and Random Forest (Scenario 1)

		Similarity			Process			Auto			All		
Project	Profile	P[1]	R[2]	F_2[3]	P	R	F_2	P	R	F_2	P	R	F_2
Derby	Bug	0.22	0.66	0.47	0.19	0.56	0.41	0.28	0.84	0.60	0.30	0.88	**0.63**
	Improvement	0.23	0.69	0.49	0.29	0.87	0.62	0.31	0.92	0.66	0.32	0.95	**0.68**
Drools	Bug	0.27	0.80	0.58	0.33	0.97	0.70	0.34	1.00	**0.72**	0.34	1.00	**0.72**
	Improvement	0.44	0.97	0.78	0.44	0.97	0.78	0.46	1.00	**0.81**	0.46	1.00	**0.81**
Groovy	Bug	0.22	0.66	0.47	0.31	0.90	0.65	0.30	0.87	0.63	0.31	0.92	**0.66**
	Improvement	0.30	0.87	0.63	0.29	0.83	0.60	0.30	0.88	0.64	0.33	0.95	**0.69**
Infinispan	Bug	0.27	0.81	0.58	0.30	0.91	0.65	0.30	0.91	0.65	0.31	0.92	**0.66**
	Improvement	0.31	0.92	0.66	0.33	0.96	0.69	0.33	0.98	**0.71**	0.33	0.98	**0.71**
Maven	Bug	0.29	0.84	0.61	0.34	0.98	0.71	0.33	0.95	0.69	0.34	0.99	**0.72**
	Improvement	0.31	0.89	0.65	0.31	0.89	0.65	0.34	0.97	**0.70**	0.33	0.94	0.68
Pig	Bug	0.30	0.90	0.64	0.32	0.97	0.69	0.33	0.98	0.70	0.33	0.99	**0.71**
	Improvement	0.32	0.94	0.68	0.34	0.99	0.71	0.34	1.00	**0.72**	0.34	1.00	**0.72**

Precision[1], Recall[2], F_2 Measure[3]

Table 5: Fully automated trace link augmentation using the different attribute sets and random forest (Scenario 2)

		Similarity			Process			Auto			All		
Project	Profile	P[1]	R[2]	$F_{0.5}$[3]	P	R	$F_{0.5}$	P	R	$F_{0.5}$	P	R	$F_{0.5}$
Derby	Bug	0.67	0.43	0.60	0.91	0.08	0.29	0.97	0.07	0.28	0.98	0.10	0.37
	Improvement	0.76	0.45	0.67	0.90	0.27	0.62	0.96	0.35	0.71	0.98	0.28	0.66
Drools	Bug	0.89	0.36	0.69	0.93	0.69	0.87	0.94	0.73	0.89	0.94	0.67	0.87
	Improvement	0.90	0.57	0.81	1.00	0.33	0.71	0.97	0.03	0.15	1.00	0.23	0.60
Groovy	Bug	0.61	0.34	0.53	0.81	0.57	0.75	0.97	0.05	0.22	0.89	0.41	0.72
	Improvement	0.70	0.56	0.67	0.88	0.44	0.73	0.85	0.63	0.80	0.90	0.58	0.81
Infinispan	Bug	1.00	0.00	0.00	0.85	0.63	0.79	0.89	0.58	0.80	0.93	0.48	0.78
	Improvement	0.92	0.66	0.85	0.89	0.61	0.82	0.93	0.69	0.87	0.97	0.69	0.89
Maven	Bug	0.88	0.46	0.74	0.97	0.38	0.74	0.94	0.45	0.77	0.99	0.37	0.74
	Improvement	0.84	0.65	0.79	0.83	0.38	0.67	0.94	0.63	0.86	0.95	0.52	0.82
Pig	Bug	0.98	0.55	0.84	0.89	0.75	0.86	0.99	0.79	0.94	0.99	0.83	0.95
	Improvement	0.98	0.74	0.92	0.92	0.85	0.90	0.97	0.92	0.96	1.00	0.90	0.98

Precision[1], Recall[2], $F_{0.5}$ Measure[3]

the second scenario is evaluated in terms of precision, recall, and $F_{0.5}$measure, because the objective of this scenario is to achieve high precision. Results are reported in Table 5 for the Random Forest classifier, which performed best. A fully automated environment requires high precision, thus we defined a project-independent cut off point based on $score > 0.95$, which achieves a precision above $\geq$ 90% across all projects when using the *All* attribute set. The other classifiers, J48 and Näive Bayes, were unable to achieve the required precision. For Random Forest, the recall drops to 50% on average as a consequence of required precision and thus only one out of two known links would be re-created. In project Derby, the recall for *All* is 10%, and similar values for the *Process*, and the *Auto* sets are achieved. However, the attributes set only containing textual similarity attributes performs best resulting in the highest $F_{0.5}$ measure, which favors precision over recall. As in the previous evaluation scenario, structural attributes do not perform well on this project, which is further discussed in the next section.

6 CONSTRUCTING UNKNOWN LINKS

The previous experiment was designed to reconstruct known links. However, the real value of our classifier is in recommending tags for commits with no existing links. While we have strong, albeit not perfect, confidence that the explicitly linked pairs of commits and issues are correctly labeled; however the non-links constitute a combination of true negative links (i.e. correctly labeled non-links) and false negative links (i.e. incorrectly labeled non-links). Of these, the false negatives represent the *missing links* that we now target. These missing links result from cases in which a developer failed to associate a commit with an issue or created an incomplete set of tags. Previous studies have reported the difficulty of correctly classifying entities not represented in the original training set [47] and we therefore need to evaluate the ability of the classifier to detect previously missing links.

Since no answer set for the non-linked commits is available, we needed to perform a manual inspection of the proposed links. As a sanity check we first evaluated whether it would be plausible to classify links on these unknown parts. In all six projects commits with links are typically related to only one issue or to a very small number of them (see Table 2). Therefore, we count the average number of issues classified as links for each of the commits without any explicit link (see Table 6). The classifier trained using attribute set *All* identifies an average of 1.54 issues per commit as a candidate link. Table 1 and 2 characterize the current linking situation in the studied projects. Based on these values, we expect a value of $\approx$ 1.2 links per commit. For example in project Infinispan, there were 2,223 commits linked to bugs and 1714 + 254 = 1968 bugs linked to commits, and thus $\approx$ 1.12 commits per bug. The ratio for improvements is 1.32. However, the classifier proposes 0.69 bugs per

Table 6: Average number of classified links between issues and a non-linked commit

Project	Profile	Similarity	Process	Auto	All
Derby	Bug	8.03	15.50	6.29	4.91
	Improvement	5.90	6.75	5.85	3.65
Drools	Bug	0.90	0.49	0.42	0.37
	Improvement	0.35	0.22	0.15	0.21
Groovy	Bug	2.05	5.41	1.52	2.53
	Improvement	2.51	2.71	2.03	1.77
Infinispan	Bug	0.00	2.63	1.74	0.69
	Improvement	1.85	1.08	0.88	0.82
Maven	Bug	0.83	0.54	0.73	0.46
	Improvement	0.94	0.72	0.65	0.61
Pig	Bug	1.40	1.93	0.93	1.20
	Improvement	0.89	1.56	1.00	1.22

commit and 0.82 improvements for every non-linked commit. That means that our approach is conservative. For project Derby, our approach underperformed. The existing ratio for linked commits per bug is 1675/(273 + 1093) = 1.21, as for the other projects. But the classifier suggests 4.91. This may stem from the imbalance of non linked commits and non linked issues in the project. There are 2, 202 non linked issues and only 553 non linked commits. However, the same imbalance of non linked issues and commits also exists in project Pig, but in this context, the classifier is unaffected.

This analysis shows that except for Derby, the classified number of links is plausible. However, it is not clear whether these links are correctly classified. To accomplish this goal we manually evaluated the correctness of a random selection of new links proposed by the classifier using the following systematic process for each project. Steps 1-4 are independently performed by one researcher (data preparer), while steps 6-7 are performed collaboratively by four additional researchers (referred to as evaluators). No communication was allowed between the researcher creating the dataset and the four evaluators during this process.

Data Set Construction for Missing Links

D1 Twenty commits without any explicitly tagged issues from the original data set for a given project were randomly selected and randomly divided into two groups *A* and *B*. 70% were placed into group A and 30% into group B.

D2 For commits in group *A* the most highly ranked issue ID was selected as the candidate link, while for commits in group *B* an issue tag that was not recommended by the classifier was selected. Group *B* was added to mitigate evaluation bias and to ensure a mix of links and non-links in the evaluation set.

D3 A randomly ordered list of each commit-issue pair selected in the previous step was generated:

Human Evaluation of Proposed Links

H1 Four human evaluators worked together to classify the first five commit-issue pairs from one randomly selected project. They performed this task without any knowledge of whether the link was recommended by the classifier or not. The evaluators then worked individually to classify the next five commit-issue pairs in the list.

Table 7: Links Recommended to Commits with no Tags

Project	Profile	L[1]	NL[2]	TP[3]	FP[4]	TN[5]	FN[6]	P[7]	R[8]
Derby	Bug	7	33	7	21	12	0	0.25	1.0
	Imp	3	37	3	25	12	0	0.11	1.0
Drools	Bug	7	33	7	21	12	0	0.25	1.0
	Imp	8	32	8	20	12	0	0.28	1.0
Maven	Bug	3	37	3	25	12	0	0.11	1.0
	Imp	2	38	1	27	11	1	0.04	0.5

Links[1], Non-Links[2], True Positives[3],False Positives[4], True Negatives[5], False Negatives[6], Precision[7], Recall[8]

H2 The Fleiss kappa inter-rater agreement was computed. Fleiss's kappa assesses the likelihood of more than two raters agreeing when classifying items into a set number of categories [17]. A kappa value of 1 means that all raters are in agreement, though a value above 0.4 indicates strong agreement. Evaluators discussed results for 20 commit-issue pairs with the aim of achieving consensus in classifying the pair as having a link or not. The Fleiss kappa value for this evaluation was approximately 0.5617, demonstrating the reliability of the evaluators to agree on the link status between a commit and issue.

H3 As satisfactory inter-rater agreement was achieved, the remaining pairs of commit-issues were split amongst the evaluators and all pairs were evaluated. The decisions made by the evaluators constitutes the "answer set" of previously unknown links against which the classifier is evaluated.

Due to the labor intensive nature of this analysis, we evaluated only three projects: Derby, Drools, and Maven. Recall and precision were computed by comparing the results returned by the classifier against the manually created "answer set". Results obtained for forty commit-issue pairs for each project are summarized in Table 7.

Results indicate that all projects except one returned a recall of 100% (i.e. 100%). The exception was Maven where recall of 100% was achieved for commit to bug links, but only 50% for commit to improvement links. In this case, there were only two true links, and one of them was missed. This means that the classifier found the pair to be unconnected while the evaluator determined that a link did exist. The precision returned for each of the three projects for both bugs and improvement was lower than the precision returned in the earlier experiments with explicitly defined links. For example, in earlier experiments Derby's precision was 0.30 for bugs and 0.32 for improvements. However, these scores dropped to 0.25 and 0.11 respectively when the classifier was used to generate links for commits with no previously known issue tags. Similar trends were observed for Drools. However, precision dropped considerably for Maven returning 0.11 for bugs, but only 0.04 for improvements. A potential explanation for the poor precision result in the Maven project is the fact that a majority of the commits represent code refactoring and in many cases were not associated with any issues at all - resulting in several false positive links. This was also the case in other projects where several commits were not directly associated with any particular issue but addressed a more trivial task such as correcting a typo or adding a comment in java docs. These types of commit negatively impact overall precision.

7 RELATED WORK

The most closely related work falls under the two areas of feature location and tracing bug reports to code.

Feature location attempts to identify sections of source code related to a specific requirement or issue. Several authors have looked at static approaches based on information retrieval techniques. For example, Antoniol et al. [1] used a probabilistic approach to retrieve trace links between code and documentation. Hayes et al. used the Vector Space Model (VSM) algorithm in conjunction with a thesaurus to establish trace links [29]. Other studies applied Latent Semantic Indexing [12, 54], Latent Dirichlet Allocation (LDA) [3, 13], or recurrent neural networks [23] to integrate semantics or context in which various terms are used. Other researchers have combined results of individual algorithms [13, 19, 37], applied AI swarm techniques [62] and combined heuristic rules with trace retrieval techniques [10, 24, 60]. Our approach leverages information retrieval to compute similarity between various types of issues, commit messages, and code. We investigated the use of LSI but rejected it for a pragmatic reason that it had a long execution time, and further, that prior studies have not shown it to outperform VSM. Ultimately we adopted a VSM-based approach, that outperformed basic VSM and integrated natural language concepts.

Researchers have also integrated structural analysis of the code to support feature location [45, 46, 49]. We did not include this in our current classifier; however, we will consider it in future work. Structural analysis may be especially helpful for finding additional classes that are related to an issue or bug. In less closely related work, researchers investigated the use of dynamic analysis for feature location [33–35]. Furthermore, Eisenbarth et al., [15] presented a technique combining dynamic and static analyses to rapidly focus on the system's parts that relate to a specific set of features.

Our work focuses not only on feature requests (i.e. improvements), but also tracing bugs to code. Canfora et al. used information retrieval techniques to identify files that were created or changed, in response to a Bugzilla ticket [6]. They identified files changed in response to similar bug reports in the past, using standard information retrieval techniques. Kim et al. predicted which source code files would change as a result of bug-fix requests [31] using Mozilla FireFox and Core code repositories as their corpus in tandem with the public Bugzilla database. They first trained a classifier to recognize 'usable' versus 'non-usable' bug reports, and then using the bugs classified as usable, trained a second classifier to identify impacted classes. Our approach differs from their work in that our goal is to generate links directly from commits to issues so that we can make direct recommendations to users if they forget to tag a commit. Our goal is therefore to create trace links as the commits are made so that developers can accept or reject them in order to create a set of trusted links. In [57], the authors proposed two heuristics, Loners and Phantoms, to infer trace links between commits and issues. We incorporate their concepts as one attribute in our classifier.

8 THREATS TO VALIDITY

There are several potential threats to the validity of our study.

Internal Validity We split the available data set for each project into 80–20% of the issues retaining the temporal ordering of the project. Choosing another split point may produce different evaluation results. We considered explicitly only "resolved" bugs and improvements, assuming that all required source code modifications had already taken place. It may be possible that the process of resolving an issue does not manifest in commits. We tried to mitigate this, by focusing on commits marked as "Fixed" or "Resolved"; however, some commits might intentionally not address an issue due to their triviality. This was evidenced in our final experiment, where our classifier recommended links even though no links existed. Furthermore, our study focused on improvements and bugs, as these were the predominant types of instances in our projects; however, we observed comparable commit link patterns for other issues types, suggesting that our approach would generalize.

External Validity Our study focused solely on open-source projects. A potential threat to external validity arises when we want to generalize our findings to a wider set of project, including commercial development. We have observed evidence of similar tagging practices in our own industrial collaborations, and therefore expect similar results. However internal company regularities might influence commit practices, and thus the overall applicability of our approach is an open question. Another threat that might limit the generalizability of our results is the use of only one combination of issue tracking system (Jira) and version control system (Git). Other tools and platforms might encourage and/or provide different linking behavior.

9 CONCLUSION

In this paper, we studied the interlinking of commits and issues in open source development projects. An analysis of six large projects showed that on average only 60% of the commits are linked to issues. This incomplete linkage fundamentally limits the establishment of project-wide traceability. To overcome this problem, we propose an approach that trains a classifier to recommend links at the time commits are made and also augments an existing set of commits and issues with automatically identified links. We identified structural, temporal, stakeholder-related and textual similarity factors as relevant information for automating this task and derived 18 attributes to quantify the relation between commit–issue pairs. A Random Forest classifier performed best on the trained attributes. We evaluated this trained model through conducting four different experiments. Two experiments studied classification performance for recommending links upon a new commit as well as for automatically augmenting missing links. We found that the classifier yielded on average 96% recall in a short list of three recommendations and could on average automatically augment every second link correctly with an average error of 4%. Finally, we manually constructed a small answer set of links from the set of previously unlinked commits and showed that the classifier returned high recall results averaging 91.6% and precision of 17.3%.

ACKNOWLEDGMENTS

The work was partially funded by the German Ministry of Education and Research (BMBF) grants: 01IS14026A, 01IS16003B, by DFG grant: MA 5030/3-1, and by the EU EFRE/Thüringer Aufbaubank (TAB) grant: 2015FE9033. It was also funded by the US National Science Foundation Grant CCF:1319680.

REFERENCES

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering* 28, 10 (2002), 970–983. https://doi.org/10.1109/TSE.2002.1041053

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug?. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 361–370. https://doi.org/10.1145/1134336

[3] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. 2010. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 95–104. https://doi.org/10.1145/1806799.1806817

[4] Adrian Bachmann and Abraham Bernstein. 2009. Software Process Data Quality and Characteristics: A Historical View on Open and Closed Source Projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops (IWPSE-Evol '09)*. ACM, 119–128. https://doi.org/10.1145/1595808.1595830

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] Gerardo Canfora and Luigi Cerulo. 2005. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium*. IEEE, 9–pp.

[7] W Cavnar. 1995. Using an n-gram-based document representation with a vector processing retrieval model. *NIST SPECIAL PUBLICATION SP* (1995), 269–269.

[8] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 55–69. https://doi.org/10.1145/2593882.2593891

[9] Jane Cleland-Huang, Mats Heimdahl, Jane Huffman Hayes, Robyn Lutz, and Patrick Mäder. 2012. Trace Queries for Safety Requirements in High Assurance Systems. In *18th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ) (LNCS)*, Vol. 7195. 179–193.

[10] Jane Cleland-Huang, Patrick Mäder, Mehdi Mirakhorli, and Sorawit Amornborvornwong. 2012. Breaking the big-bang practice of traceability: Pushing timely trace recommendations to project stakeholders. In *2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, IL, USA, September 24-28, 2012*, Mats Per Erik Heimdahl and Pete Sawyer (Eds.). IEEE Computer Society, 231–240. https://doi.org/10.1109/RE.2012.6345809

[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4-5 (2012), 531–577.

[12] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. 2004. Enhancing an Artefact Management System with Traceability Recovery Features. In *20th IEEE International Conference on Software Maintenance (ICSM)*. 306–315.

[13] Alex Dekhtyar, Jane Huffman Hayes, Senthil Karthikeyan Sundaram, Elizabeth Ashlee Holbrook, and Olga Dekhtyar. 2007. Technique Integration for Requirements Assessment. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*. IEEE Computer Society, 141–150. https://doi.org/10.1109/RE.2007.17

[14] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, René L. Krikhaar, Ralf Lämmel, and Chris Verhoef (Eds.). IEEE Computer Society, 53–62. https://doi.org/10.1109/ICPC.2008.39

[15] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating features in source code. *IEEE Transactions on software engineering* 29, 3 (2003), 210–224.

[16] Davide Falessi, Massimiliano Di Penta, G. Canfora, and G. Cantone. to appear. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering* (to appear).

[17] Joseph L. Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin* 76, 5 (1971), 378 – 382.

[18] Apache Software Foundation. [n. d.]. How Should I Apply Patches From A Contributor. ([n. d.]). http://www.apache.org/dev/committers.html#applying-patches

[19] M. Gethers, R. Oliveto, D. Poshyvanyk, and Andrea De Lucia. 2011. On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Link Recovery. In *27th IEEE International Conference on Software Maintenance (ICSM)*. 133–142.

[20] Git SCM [n. d.]. Git SCM. ([n. d.]). http://www.git-scm.com.

[21] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. 2012. Traceability Fundamentals. In *Software and Systems Traceability*, Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman (Eds.). Springer, 3–22. http://dx.doi.org/10.1007/978-1-4471-2239-51 10.1007/978-1-4471-2239-51.

[22] O. C. Z. Gotel and Anthony Finkelstein. 1994. An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering, ICRE '94, Colorado Springs, Colorado, USA, April 18-21, 1994*. IEEE, 94–101. https://doi.org/10.1109/ICRE.1994.292398

[23] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 3–14. https://doi.org/10.1109/ICSE.2017.9

[24] Jin Guo, Jane Cleland-Huang, and Brian Berenbach. 2013. Foundations for an expert system in domain-specific traceability. In *21st IEEE International Requirements Engineering Conference (RE)*. 42–51. https://doi.org/10.1109/RE.2013.6636704

[25] Jin Guo, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin, Jane Huffman Hayes, and Michael Vierhauser. 2016. Cold-start software analytics. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. 142–153. https://doi.org/10.1145/2901739.2901740

[26] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust Prediction of Fault-Proneness by Random Forests. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, Washington, DC, USA, 417–428. https://doi.org/10.1109/ISSRE.2004.35

[27] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278

[28] M. A. Hall. 1998. *Correlation-based Feature Subset Selection for Machine Learning*. Ph.D. Dissertation. University of Waikato, Hamilton, New Zealand.

[29] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Transactions on Software Engineering* 32, 1 (2006), 4–19.

[30] JIRA [n. d.]. Jira Issue Tracking Software. ([n. d.]). http://www.jira.com.

[31] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *Software Engineering, IEEE Transactions on* 39, 11 (2013), 1597–1610.

[32] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with Noise in Defect Prediction. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 481–490. https://doi.org/10.1145/1985793.1985859

[33] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. 2015. Can method data dependencies support the assessment of traceability between requirements and source code? *Journal of Software: Evolution and Process* 27, 11 (2015), 838–866. https://doi.org/10.1002/smr.1736

[34] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lv, and Alexander Egyed. 2012. Do data dependencies in source code complement call dependencies for understanding requirements traceability?. In *28th IEEE International Conference on Software Maintenance (ICSM)*. 181–190.

[35] Hongyu Kuang, Jia Nie, Hao Hu, Patrick Rempel, Jian Lu, Alexander Egyed, and Patrick Mäder. 2017. Analyzing closeness of code dependencies for improving IR-based Traceability Recovery. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 68–78. https://doi.org/10.1109/SANER.2017.7884610

[36] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. Improving trace accuracy through data-driven configuration and composition of tracing features. In *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 378–388.

[37] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. Improving trace accuracy through data-driven configuration and composition of tracing features. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 378–388.

[38] Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvanyk. 2012. Information Retrieval Methods for Automated Traceability Recovery. In *Software and Systems Traceability*. 71–98. https://doi.org/10.1007/978-1-4471-2239-5_4

[39] Patrick Mäder and Jane Cleland-Huang. 2013. A visual language for modeling and executing traceability queries. *Software and System Modeling* 12, 3 (2013), 537–553.

[40] Patrick Mäder and Jane Cleland-Huang. 2015. From Raw Project Data to Business Intelligence. *IEEE Software* 32, 4 (2015), 22–25. https://doi.org/10.1109/MS.2015.92

[41] Patrick Mäder and Alexander Egyed. 2015. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering* 20, 2 (2015), 413–441. https://doi.org/10.1007/s10664-014-9314-z

[42] Patrick Mäder, Paul L. Jones, Yi Zhang, and Jane Cleland-Huang. 2013. Strategic Traceability for Safety-Critical Projects. *IEEE Software* 30, 3 (2013), 58–66.

[43] Patrick Mäder, Rocco Oliveto, and Andrian Marcus. 2017. Empirical studies in software and systems traceability. *Empirical Software Engineering* 22, 3 (2017), 963–966. https://doi.org/10.1007/s10664-017-9509-1

[44] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of*

mathematical statistics (1947), 50–60.

[45] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending source code for use in rapid software prototypes. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 848–858. https://doi.org/10.1109/ICSE.2012.6227134

[46] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, Anthony Cleve, Filippo Ricca, and Maura Cerioli (Eds.). IEEE Computer Society, 199–208. https://doi.org/10.1109/CSMR.2013.29

[47] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, Michele Lanza, Massimiliano Di Penta, and Tao Xie (Eds.). IEEE Computer Society, 40–49. https://doi.org/10.1109/MSR.2012.6224298

[48] Balasubramaniam Ramesh and Matthias Jarke. 2001. Toward Reference Models of Requirements Traceability. *IEEE Transactions on Software Engineering* 27, 1 (2001), 58–93.

[49] Ghulam Rasool and Patrick Mäder. 2011. Flexible design pattern detection based on feature types. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 243–252. https://doi.org/10.1109/ASE.2011.6100060

[50] Michael Rath, Jacob Rendall, Jin Guo, Jane Cleland-Haung, and Patrick Mäder. 2018. Replication Data for: Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. https://goo.gl/2NMjcF. (2018). https://doi.org/10.7910/DVN/BZLSLA

[51] Gilbert Regan, Miklós Biró, Fergal McCaffery, Kevin McDaid, and Derek Flood. 2014. A Traceability Process Assessment Model for the Medical Device Domain. In *Systems, Software and Services Process Improvement - 21st European Conference, EuroSPI 2014, Luxembourg, June 25-27, 2014. Proceedings (Communications in Computer and Information Science)*, Béatrix Barafort, Rory V. O'Connor, Alexander Poth, and Richard Messnarz (Eds.), Vol. 425. Springer, 206–216. https://doi.org/10.1007/978-3-662-43896-1_18

[52] Patrick Rempel and Patrick Mäder. 2016. A quality model for the systematic assessment of requirements traceability. In *Software Engineering 2016, Fachtagung des GI-Fachbereichs Softwaretechnik, 23.-26. Februar 2016, Wien, Österreich (LNI)*, Jens Knoop and Uwe Zdun (Eds.), Vol. 252. GI, 37–38. http://subs.emis.de/LNI/Proceedings/Proceedings252/article52.html

[53] Patrick Rempel and Patrick Mäder. 2017. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Trans. Software Eng.* 43, 8 (2017), 777–797. https://doi.org/10.1109/TSE.2016.2622264

[54] Patrick Rempel, Patrick Mäder, and Tobias Kuschke. 2013. Towards feature-aware retrieval of refinement traces. In *7th International Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE 2013, 19 May, 2013, San Francisco, CA, USA*, Nan Niu and Patrick Mäder (Eds.). IEEE Computer Society, 100–104. https://doi.org/10.1109/TEFSE.2013.6620163

[55] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. 2014. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 943–954. https://doi.org/10.1145/2568225.2568290

[56] Bilyaminu Auwal Romo, Andrea Capiluppi, and Tracy Hall. 2014. Filling the Gaps of Development Logs and Bug Issue Data. In *Proceedings of The International Symposium on Open Collaboration, OpenSym 2014, Berlin, Germany, August 27 - 29, 2014*, Dirk Riehle, Jesús M. González-Barahona, Gregorio Robles, Kathrin M. Möslein, Ina Schieferdecker, Ulrike Cress, Astrid Wichmann, Brent J. Hecht, and Nicolas Jullien (Eds.). ACM, 8:1–8:4. https://doi.org/10.1145/2641580.2641592

[57] Gerald Schermann, Martin Brandtner, Sebastiano Panichella, Philipp Leitner, and Harald C. Gall. 2015. Discovering loners and phantoms in commit and issue data. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, Andrea De Lucia, Christian Bird, and Rocco Oliveto (Eds.). IEEE Computer Society, 4–14. https://doi.org/10.1109/ICPC.2015.10

[58] Marcus Seiler and Barbara Paech. 2017. Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems - A Research Preview. In *Requirements Engineering: Foundation for Software Quality - 23rd International Working Conference, REFSQ 2017, Essen, Germany, February 27 - March 2, 2017, Proceedings (Lecture Notes in Computer Science)*, Paul Grünbacher and Anna Perini (Eds.), Vol. 10153. Springer, 174–180. https://doi.org/10.1007/978-3-319-54045-0_13

[59] Yonghee Shin, Jane Huffman Hayes, and Jane Cleland-Huang. 2015. Guidelines for Benchmarking Automated Software Traceability Techniques. In *8th IEEE/ACM International Symposium on Software and Systems Traceability, SST 2015, Florence, Italy, May 17, 2015*. 61–67. https://doi.org/10.1109/SST.2015.13

[60] George Spanoudakis, Andrea Zisman, Elena Pérez-Miñana, and Paul Krause. 2004. Rule-based generation of requirements traceability relations. *Journal of Systems and Software* 72, 2 (2004), 105–127.

[61] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. 2017. Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework. *Empirical Software Engineering* 22, 3 (2017), 967–995. https://doi.org/10.1007/s10664-016-9457-1

[62] Hakim Sultanov, Jane Huffman Hayes, and Wei-Keat Kong. 2011. Application of swarm techniques to requirements tracing. *Requirements Engineering* 16, 3 (2011), 209–226.

[63] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.