Extracting and Comparing Concepts Emerging from Software Code, Documentation and Tests

Zaki Pauzi[1], Andrea Capiluppi[1]

[1]*Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence (University of Groningen), Nijenborgh 9, 9747 AG Groningen, The Netherlands*

Abstract

Traceability in software engineering is the ability to connect different artifacts that have been built or designed at various points in time. Given the variety of tasks, tools and formats in the software lifecycle, an outstanding challenge for traceability studies is to deal with the heterogeneity of the artifacts, the links between them and the means to extract each. Using a unified approach for extracting keywords from textual information, this paper aims to compare the concepts extracted from three software artifacts: source code, documentation and tests from the same system. The objectives are to detect similarities in the concepts emerged, and to show the degree of alignment and synchronisation the artifacts possess. Using the components of three projects from the Apache Software Foundation, this paper extracts the concepts from 'base' source code, documentation, and tests (separated from the source code). The extraction is done based on the keywords present in each artifact: we then run multiple comparisons (through calculating cosine similarities on features extracted by word embeddings) in order to detect how the sets of concepts are similar or overlap. For similarities between code and tests, we discovered that using pre-trained language models (with increasing dimension and corpus size) correlates to the increase in magnitude, with higher averages and smaller ranges. FastText pre-trained embeddings scored the highest average of 97.33% with the lowest range of 21.8 across all projects. Also, our approach was able to quickly detect outliers, possibly indicating drifts in traceability within modules. For similarities involving documentation, there was a considerable drop in similarity score compared to between code and tests per module – down to below 5%.

Keywords

software traceability, natural language processing, information retrieval, textual analysis

1. Introduction

Software traceability is a fundamentally important task in software engineering: for some domains, traceability is even assessed by certifying bodies [1]. The need for automated traceability increases as projects become more complex and as the number of artifacts increases [2, 3, 4]. The underlying complexities of the logical relations between these artifacts have prompted a variety of empirical studies [5, 6, 7] and several areas of research, particularly in the inception of semantic domain knowledge [8, 9]. Above all, one of the most pressing challenges is linked to the heterogeneity of the artifacts, the links extracted between them, and the variety of formats and tools available for the different stages of traceability studies [10]. Much has been done in

BENEVOL'21: The 20th Belgium-Netherlands Software Evolution Workshop, December 07–08, 2021, 's-Hertogenbosch (virtual), NL
a.z.bin.mohamad.pauzi@rug.nl (Z. Pauzi); a.capiluppi@rug.nl (A. Capiluppi)
0000-0003-4032-4766 (Z. Pauzi); 0000-0001-9469-6050 (A. Capiluppi)

re-establishing traceability links between software artifacts, but not particularly in the field of *domain concepts* by applying semantic modelling. Major advancement in NLP research in recent years has resulted in practical uses of language models [11], such as the introduction of deep and transfer learning, which was more commonly used in computer vision. By applying this to software artifacts, the human language can be closely related to concepts extracted from source code and tests, ultimately comprehending the software's identity through natural language. This paves the way for a variety of semantics-driven applications, such as automated software domain classification, taxonomy structure of domains and studies of software evolution.

The contribution of this paper is the notion of *concept similarity*, in the context of the traceability between source code, documentation and tests of a software unit (class, module or system). The book definition of "concept" is a principle or an idea [1] that is abstract or generic in nature [2]. In software engineering, we add on to this definition to include the features in software components, where *concepts* are presently *identified* through source code analysis and manipulation. For our dataset, we use the modules present in three actively managed Java open source projects from the Apache Software Foundation (ASF): Apache's Dubbo[3], Skywalking[4] and Flink[5]. Although the sample is small, our aim is to showcase the methodology behind the data extraction: the approach is straightforward and scalable, so it will be possible to analyse a larger sample with minimal effort. We extracted the concepts of each project from three sources: the source code, documentation and tests. The documentation was extracted from the README file, which serves as the first *point of entry* for project stakeholders. The tests of each project were identified through regular expressions of filenames, while the remaining Java files constitute the 'base' source code for these projects. We extracted the concepts emerging from the keywords used in each of these sources, and we run multiple similarity measurements to determine the similarity of these artifacts, answering the following:

RQ1: How *similar* (syntactically and semantically) are the three software artifacts: source code, documentation and tests?

RQ2: How does feature extraction (through word embeddings) perform when comparing textual similarity between the source code, documentation and tests?

This paper is structured as follows: Section 2 summarises the related work, and Section 3 describes the unified approach used to extract the concepts. Section 4 shows the results and discusses the findings. Section 5 concludes.

2. Background Research

In [12], the authors present a *roadmap* of the research on software traceability, based on the topics that researchers had focused on so far. According to that roadmap, our work:

- is based on *vertical* traceability relations (i.e., it includes the relations between different artifacts);

[1]https://dictionary.cambridge.org/dictionary/english/concept
[2]https://www.merriam-webster.com/dictionary/concept
[3]https://github.com/apache/dubbo
[4]https://github.com/apache/skywalking
[5]https://github.com/apache/flink

- includes *overlap* relations (i.e., we study whether C1 and C2 [6] refer to common concepts of a system);
- targets the *automatic generation* of traceability relations (as compared to manual or semi-automatic approaches);
- aims to analyse traceability for the purpose of software validation, verification and testing, insofar as '*traceability relations may be used to check the existence of appropriate test cases for verifying different requirements and to retrieve such cases for further testing*' [12].

The traceability field is most often associated with requirements traceability [13]: pre-packaged, automated tools like TraceLab [14] have often been preferred to trace different artifacts, or versions of the same artifact. Although it became clear over the years that traceability of software artifacts is essentially an information retrieval problem [15], combining software traceability with semantic information of software artifacts was shown to be a promising technique. Prospective traceability tools [16] have been developed with both the architecture and semantic modelling in mind. This empirical approach to recover traceability links through topic modelling was later adapted and improved through integrating orthogonal information retrieval methods [17].

Whilst most of the traditional literature on traceability has focused on requirements, the research on traceability of open source systems has focused on other artifacts (source code, user documents, build management documents etc [18, 19]). When requirements are considered, they are not traditionally elicited through customer feedback, but just-in-time and termed *feature requests* [20], or elicited using various other artifacts (e.g., CVS/SVN commit logs and Bugzilla reports [21]). In the open source context, the extraction of requirements is often considered as a long term view, for instance in the context of impact analysis [22].

In the context of deriving semantic value from software artifacts, extracting topics from source code has been previously presented in [23] by demonstrating that the Latent Dirichlet Allocation (LDA) technique has a strong applicability in the extraction of topics from classes, packages and overall systems. This technique was also used in a later paper [24] where experts were consulted to assign software system in accordance to its domain. The keyword terms derived from this technique were also compared in terms of text similarity using various word embeddings in [25].

3. Methodology

3.1. Definitions

The following are the definitions as used in this paper:

- **Corpus keyword** (term) – given the source code contained in a class, a token or term is any item that is contained in the source code. We do not consider as a term any of the Java-specific keywords (e.g., `if`, `then`, `switch`, etc.). Additionally, the camelCase or PascalCase notations are first decoupled in their components (e.g., the class constructor *InvalidRequestTest* produces the terms *invalid*, *request* and *test*).

[6]"C1" and "C2" are arbitrary concepts

- **Topic** – this refers to the clusters of keywords extracted with the Latent Dirichlet Allocation (LDA) technique, and weighted by their relative relevance. For this paper, we will concatenate all the keywords sans weights as *topic keywords*.
- **Concept** – the set of 'source code concepts' is the union of the (i) corpus keywords set and (ii) topics set, as extracted from the source code. These 'concepts' are derived from the lexicon used in the code. The 'test concepts' have a similar definition, using the sets from the test batch.
- x/y **SIM** – Concept similarity between x and y, where x and y are software artifacts, and $x \neq y$.

3.2. Selection of Software Systems

Table 1 shows the details of our sample dataset. We are fully aware that the analysis of three systems (instead of hundreds, or thousands) does not allow to draw any conclusion for any other system. The empirical study that we present below focuses on top rated systems (representing the quality of code developed by the ASF community, adhering strictly to established coding standards[7]) rather than promoting the width of representativeness. We will further discuss about the implications of our choice in the threats to external validity.

Table 1
Selection of Software Systems

ID	Project Name	Modules	Stars
P1	Apache Dubbo	14	36,323
P2	Apache Skywalking	4	17,994
P3	Apache Flink	27	17,378

3.3. Concept Extraction

The extraction of the concepts and similarity measurements was executed in Python via a Jupyter notebook[8]. The extraction was carried out to build the class corpus for each project. Results were then compared and analysed.

For any project's **source code** and **tests**, we extracted all the class names and identifiers that were used for methods and attributes. Additionally, inline comments were extracted as well, but the Java keywords were excluded[9], along with the project names (such as `apache` and `dubbo` for P1) to provide a more accurate representation. This results in an extraction that comprehensively represents the semantic overview of concepts whilst minimising noise from code syntax. The final part of the extraction is the lemmatisation of the terms using SpaCy's[10] `token.lemma_`. Lemmatising is deriving the *base* from the terms (also called their dictionary form, or 'lemma'), thus enabling more matches when we compare from the different

[7]https://directory.apache.org/fortress/coding-standards.html
[8]https://github.com/zakipauzi/benevol2021/blob/main/concept_similarity_benevol.ipynb
[9]https://en.wikipedia.org/wiki/List_of_Java_keywords
[10]https://spacy.io

sources (e.g., `best -> good`). An excerpt of the complete corpus from the source code of P1's `dubbo-common` module is shown at figure 1.

Extending this to all the modules in the three projects, we look at the textual similarity of corpus keywords extracted for (i) source code and (ii) tests per module.

```
activate reference service metadata colon separator service key service version
registry config config available serial...
```

Figure 1: Excerpt from the corpus of `dubbo-common` base code from P1

For the **documentation** extraction, we looked at the README file and ran through a similar cleaning pipeline. All non-English characters were disregarded during the exercise by checking if the character falls within the ASCII Latin space. Figure 2 shows a simplified diagram of our concept similarity (i.e., traceability) between artifacts.

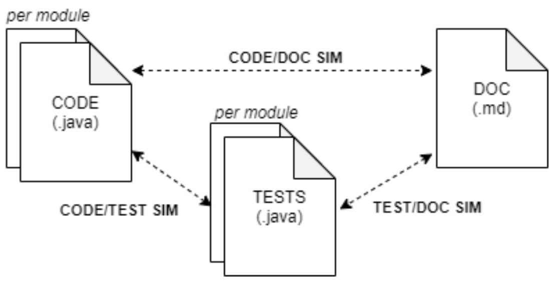

Figure 2: Diagram of concept similarity between artifacts

3.4. Topic Modelling

For the CODE/DOC SIM and TEST/DOC SIM, we run topic modelling with Latent Dirichlet Allocation (LDA) due to the vast difference in token count between the README file and the code. We cluster the extracted class corpus into groups; an unsupervised learning approach to tag groups of terms to a topic based on their semantic context, then identifying the overarching theme of each cluster through their topics. Using *Gensim*[11] LDA, we identify the topic clusters present. Next, we concatenate all the topic keywords from all the modules from code and tests respectively. Figure 3 shows an example of topic keywords that emerge from P1's `dubbo-common` module source code.

These are used to compare with the topic keywords emerging from documentation. In Section 4, we will look at the results of proportion in overlap of these concepts extracted between code and documentation, and tests and documentation.

[11] https://radimrehurek.com/gensim

```
'config', 'logger', 'key', 'path', 'level', 'listener', 'msg', 'throwable',
'stream', 'provider', 'map', 'integer', 'object', ...
```

Figure 3: dubbo-common's source code topic keywords

3.5. Similarity Measures

We apply multiple vectorisation techniques to represent corpus keywords as vectors in a vector space, and then we run cosine similarity against these vectors to address **RQ1** and **RQ2**. Cosine similarity is a "distance" metric, irrespective of orientation and magnitude: the lower the angle between the vectors, the higher the similarity. Table 2 shows the different vectorisation techniques used with cosine similarity for measurement.

Table 2
Vectorisation Techniques with Cosine Similarity

Vectorisation type	Pre-trained?	Dimensions	Source
TFIDF Vectorizer	No	-	scikit-learn [26]
SO W2V	Yes	200	Online at [27]
SpaCy	Yes	300	Online at [28]
FastText	Yes	300	Online at [29]

A toolchain graph showing each part of the process discussed above (subsections IIIA. to IIIE.) is shown in Figure 4.

4. Results

For space limitations we have made available online the result summary of Code and Test similarity (CODE/TEST SIM)[12], and the results of Code and Documentation similarity (CODE/DOC SIM) and Test and Documentation similarity (TEST/DOC SIM)[13] across all modules in each project.

Figure 5 shows the box plot distribution of CODE/TEST SIM for the vectorisation techniques across all modules for P1, P2 and P3 (**RQ2**). Similarly to the trends found in [25], the analysis of the three Apache projects confirms that the baseline TF-IDF measurement has the widest ranges of 0.62 to 0.85, and lowest mean scores of 0.34 to 0.66. At the same time, the role of pre-trained embeddings is central in deriving semantic context to concepts: contextual similarity (helped by the three trained datasets) is linked to a higher cosine similarity when comparing corpus keywords in software artifacts. This is expected, since the Java syntax closely mirrors the words already present in the pre-trained vector spaces.

With pre-trained embeddings trained on a wide vector space (e.g, an English vocabulary across all domains), the range of similarity score gets even narrower. We see this range decrease

[12]https://github.com/zakipauzi/benevol2021/blob/main/benevolcodetestsimsummary.csv
[13]https://github.com/zakipauzi/benevol2021/blob/main/benevolcodetestdocsimsummary.csv

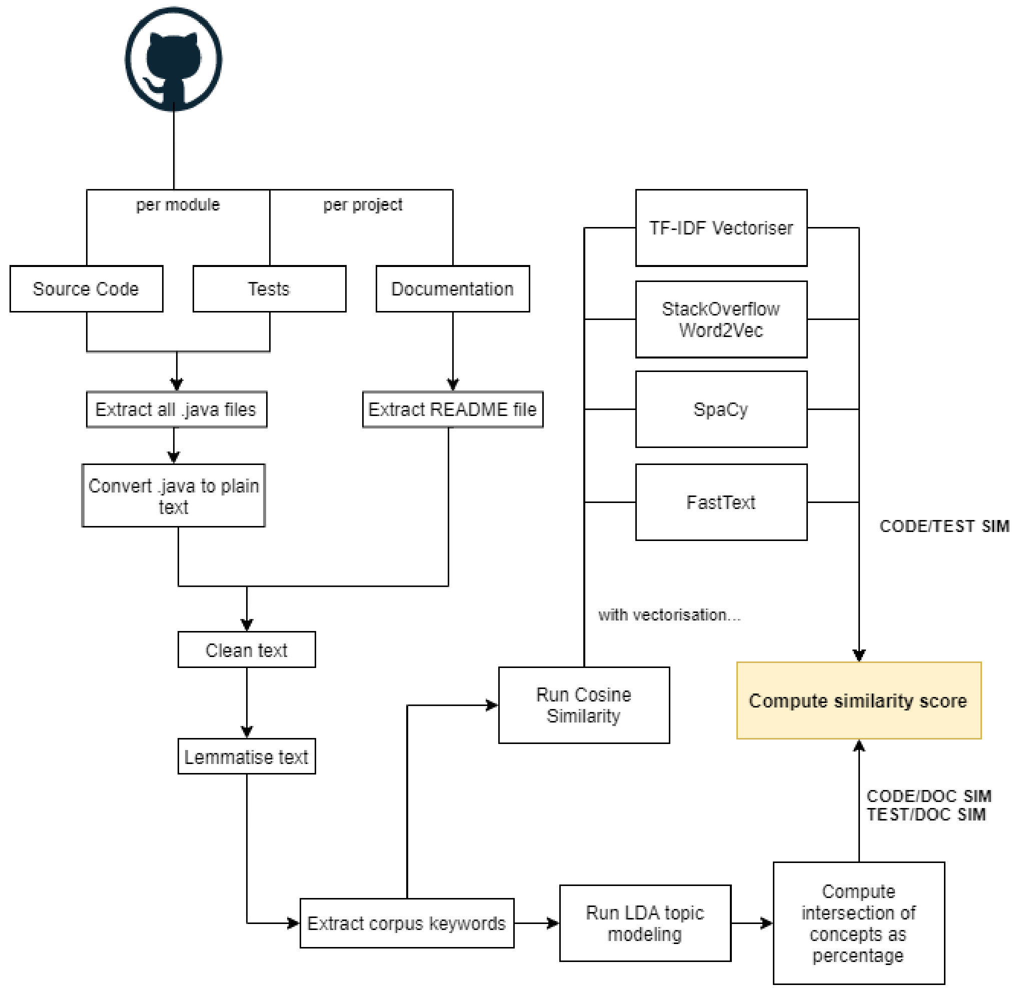

Figure 4: Toolchain implemented throughout the paper

as we run CODE/TEST SIM with StackOverflow data (SO W2V), expanding vocabulary beyond just within the scope of our current artifacts (TF-IDF), and further decrease with SpaCy and FastText embeddings. In other words, the wider the vector space of embeddings, the narrower the range and the higher the mean. Ultimately, we know that a high score does not necessarily denote a *better similarity*, but it is worth exploring further on how we can continue to balance the role of pre-trained embeddings with accurately representing similarity that is domain-specific, which ideally relates to our traceability reconstruction solution through concept similarities.

A key advantage of using this approach (particularly with pre-trained embeddings) is that the outliers can be easily detected in figure 5, indicating drifts in traceability via similarity measurements. For P1, we can see outliers such as modules `dubbo-container`, `dubbo-filter` and P3's `flink-streaming-scala`: these show that the concepts emerging from their code and tests are vastly different and need looking into.

As for our similarity measurements: CODE/DOC SIM and TEST/DOC SIM, comparing module

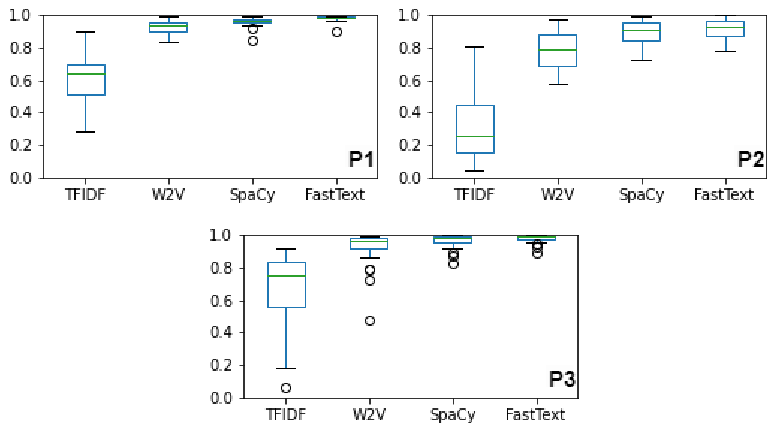

Figure 5: CODE/TEST similarity for the modules contained in each system

to the documentation in its entirety will not be accurate, thus we ran topic modelling to represent the hierarchical structure of the syntax more effectively, a similar approach to [24]. Topics emerging from corpus keywords represent clusters that are a level higher than the corpus keywords, bridging the gap between per module syntax with README corpus. The results for both similarities (using Jaccard Index on topic keywords) are well below CODE/TEST SIM: >5%. Further work will need to be done to establish the accuracy of this result, such as incorporating weights to topics and identifying traceability beyond the syntax of keywords. across P1 modules scored lower than baseline whereas it differs for some P2 modules.

4.1. Threats to validity

The approach that we have shown has some threats and limitations that we have identified.

- *External tests and documentation not included.* Other than the basic unit and logic tests, we do have other tests that may involve other software and systems, which are not included (i.e., integration tests).
- *Construct: definition of concepts as a construct.* Our definition looked at both facets – the corpus keywords and the derived topics. Moreover, our dataset assumes on the notion that top rated projects from an open source body with established coding standards (such as ASF) represent *good* code quality in artifacts and hence, we expect that the concepts emerging from the sources are aligned, as in [25]. Our approach ensures that artifacts are treated similarly, and *drifts* between artifacts are clearly captured by outliers.
- *Conclusion: non uniform identifiers and code smells.* From the analysis of the systems, we observed that the structure of code has some degree of non-uniformity in the way identifiers are used to represent meaning.

5. Conclusion and Further Work

In this paper we explored the triangulation of textual similarity via different techniques between the concepts extracted from the source code, documentation and tests of Java software systems. The aim was to assess the traceability between the three sources, and to put it in the context of concept overlap. There is great potential in the results for further development and analysis in semantic traceability for software artifacts (e.g., establishing connections between topics derived outside of lexical intersection, exploring different metrics to detect traceability). Also, expanding our dataset to include projects of various domains, languages and categories. Moving forward, we would extend this solution to include other artifacts such as architecture diagrams, bug reports and functional requirements. We would also want to look at ways to adopt this approach to a supervised automated logic of domain categorisation.

References

[1] J. Guo, J. Cheng, J. Cleland-Huang, Semantically enhanced software traceability using deep learning techniques, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 3–14.

[2] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, E. Romanova, Best practices for automated traceability, Computer 40 (2007) 27–35.

[3] C. Duan, P. Laurent, J. Cleland-Huang, C. Kwiatkowski, Towards automated requirements prioritization and triage, Requirements engineering 14 (2009) 73–89.

[4] J. Guo, M. Gibiec, J. Cleland-Huang, Tackling the term-mismatch problem in automated trace retrieval, Empirical Software Engineering 22 (2017) 1103–1142.

[5] J. I. Maletic, E. V. Munson, A. Marcus, T. N. Nguyen, Using a hypertext model for traceability link conformance analysis, in: Proc. of the Int. Workshop on Traceability in Emerging Forms of Software Engineering, 2003, pp. 47–54.

[6] H. Schwarz, J. Ebert, A. Winter, Graph-based traceability: a comprehensive approach, Software & Systems Modeling 9 (2010) 473–492.

[7] P. Mäder, R. Olivetto, A. Marcus, Empirical studies in software and systems traceability, Empirical Softw. Engg. 22 (2017) 963–966. URL: https://doi.org/10.1007/s10664-017-9509-1. doi:10.1007/s10664-017-9509-1.

[8] A. Marcus, J. I. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, in: 25th International Conference on Software Engineering, 2003. Proceedings., IEEE, 2003, pp. 125–135.

[9] T. Zhao, Q. Cao, Q. Sun, An improved approach to traceability recovery based on word embeddings, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2017, pp. 81–89.

[10] R. M. Parizi, S. P. Lee, M. Dabbagh, Achievements and challenges in state-of-the-art software traceability between test and code artifacts, IEEE Transactions on Reliability 63 (2014) 913–926.

[11] A. Torfi, R. A. Shirvani, Y. Keneshloo, N. Tavvaf, E. Fox, Natural language processing advancements by deep learning: A survey, ArXiv abs/2003.01200 (2020).

[12] G. Spanoudakis, A. Zisman, Software traceability: a roadmap, in: Handbook Of Software Engineering And Knowledge Engineering: Vol 3: Recent Advances, World Scientific, 2005, pp. 395–428.

[13] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. A. Raja, K. Kamran, Requirements traceability: a systematic review and industry case study, International Journal of Software Engineering and Knowledge Engineering 22 (2012) 385–433.

[14] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, et al., Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 1375–1378.

[15] M. Borg, P. Runeson, A. Ardö, Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability, Empirical Software Engineering 19 (2014) 1565–1616.

[16] H. U. Asuncion, A. U. Asuncion, R. N. Taylor, Software traceability with topic modeling, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, volume 1, 2010, pp. 95–104. doi:10.1145/1806799.1806817.

[17] M. Gethers, R. Oliveto, D. Poshyvanyk, A. D. Lucia, On integrating orthogonal information retrieval methods to improve traceability recovery, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 133–142. doi:10.1109/ICSM.2011.6080780.

[18] H. Kagdi, J. I. Maletic, B. Sharif, Mining software repositories for traceability links, in: 15th IEEE International Conference on Program Comprehension (ICPC'07), IEEE, 2007, pp. 145–154.

[19] H. Kagdi, J. Maletic, Software repositories: A source for traceability links, in: International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE07), 2007, pp. 32–39.

[20] P. Heck, A. Zaidman, Horizontal traceability for just-in-time requirements: the case for open source feature requests, Journal of Software: Evolution and Process 26 (2014) 1280–1296.

[21] N. Ali, Y.-G. Guéhéneuc, G. Antoniol, Trustrace: Mining software repositories to improve the accuracy of requirement traceability links, IEEE Transactions on Software Engineering 39 (2012) 725–741.

[22] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 430–440.

[23] A. Kuhn, S. Ducasse, T. Gírba, Semantic clustering: Identifying topics in source code, Information and Software Technology 49 (2007) 230–243.

[24] A. Capiluppi, N. Ajienka, N. Ali, M. Arzoky, S. Counsell, G. Destefanis, A. Miron, B. Nagaria, R. Neykova, M. Shepperd, et al., Using the lexicon from source code to determine application domain, in: Proceedings of the Evaluation and Assessment in Software Engineering, 2020, pp. 110–119.

[25] Z. Pauzi, A. Capiluppi, Text similarity between concepts extracted from source code and documentation, in: Intelligent Data Engineering and Automated Learning – IDEAL

2020 - 21st International Conference, 2020, Proceedings, 2020, pp. 124–135. doi:10.1007/978-3-030-62362-3_12.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[27] V. Efstathiou, C. Chatzilenas, D. Spinellis, Word embeddings for the software engineering domain, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), 2018, pp. 38–41.

[28] explosion, en_core_web_md, https://github.com/explosion/spacy-models/releases/tag/en_core_web_md-3.1.0, 2021.

[29] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, A. Joulin, Advances in pre-training distributed word representations, 2017. arXiv:1712.09405.