

Evaluation of Textual Similarity Techniques in Code Level Traceability

Viktor Csuvik[1(✉)], András Kicsi[1], and László Vidács[1,2]

[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
{csuvikv,akicsi,lac}@inf.u-szeged.hu
[2] MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Szeged, Hungary

Abstract. Automatic recovery of test-to-code traceability links is an important task in many areas of software engineering, like quality assurance and code maintenance. The research community has shown great interest in such a topic and has developed several techniques that already made significant advances in the field. These techniques include text-based learning algorithms, of which corpus is built from the source code of the software components. Several techniques based on information retrieval have been benchmarked, but the capabilities of many learning algorithms have not yet been tested. In this work we examine the textual similarity measures produced by three different machine learning techniques for the recovery of traceability information while also considering various textual representations of the source code. The obtained results are evaluated on 4 open source systems based on naming conventions. We have been able to improve the current textual similarity based state-of-the-art results in the case of each evaluated system.

Keywords: Traceability · Testing · Test-to-code · Machine learning · Text similarity

1 Introduction

True perfection in software engineering does not exist. Software testing, however, constitutes a major aspect in the assurance of quality. Besides simply detecting faults in software, tests are also essential for other areas in software engineering, like Automatic Program Repair (APR), where tests are needed in the patch generation process, or code maintenance. The primary aspect of testing is to provide information on whether the software achieves the general result its stakeholders desire. Testing can provide an independent view of the software and opens new opportunities in calculating the risks. It is known that complete testing is not fully achievable, still writing tests on edge cases and increasing their amount is considered to be a good coding practice. It is not a coincidence that large systems often incorporate vast amounts of tests.

Considering tens of thousands of tests, their maintenance becomes cumbersome and the goal of some tests may even become unknown. In these cases

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 529–543, 2019.
https://doi.org/10.1007/978-3-030-24305-0_40

recovering which test case assesses a specific part of code can prove to be a challenge. Traceability in general stands for the task of tracing software items through a variety of software products. The previously described specific problem is called *test-to-code* traceability. Traceability is a well-researched area with a serious industrial background. While the most widespread problem in this field is domain requirement traceability [3,22], test-to-code traceability also gained attention from the research community [5,15,35].

Using good coding practices [41] can make the task easier and with proper naming conventions [35] very accurate results can be achieved. However, if a developer lacks these skills or proper foresight, the traceability problem becomes non-trivial with many pitfalls. In these cases, automatic recovery approaches should be introduced, which does not require such assumptions from the examined system. While several attempts have already been made to cope with this problem, these techniques are limited since they typically depend on intuitive features. In our previous work [5,15] we provided a method, that automatically links test cases and production classes relying only on conceptual information. In the current paper, we make an attempt to improve our former results by involving new machine learning techniques. We compare these results and also show that combining them outperforms the current semantic information based approaches in the traceability task.

The paper is organized as follows. We present a high-level overview of our research in the following section by depicting the proposed approach to recover test-to-code links and specify our research objectives. Next, we introduce the examined database and its representations upon which the experiments were carried out. Evaluation on four systems and analysis are presented in Sect. 5. Related work is discussed in Sect. 6, and we conclude the paper in the last section.

2 Overview

Test-to-code traceability means finding the links between test cases and production code. More precisely for a test case we want to find certain parts of the code which it was meant to test. For a large system, this task can be challenging, particularly when the development lacks good coding practices [41] like proper naming conventions. Using practices like naming the test classes after the tested production code automatically creates a conceivable link between the test and the tested artifact. It is well known, that with proper naming conventions, retrieving traceability links is a minor task [35]. If we consider, however, a system where the targets of the test cases are unknown to us, other approaches should be applied.

Figure 1 provides an illustration of the comprehensive approach we propose. We consider a software system written in the Java programming language. It contains both test classes and production classes and we aim to recover the relationship between them. We made no assumptions about the names of the software artifacts. From the raw source code, we extract the classes of the system

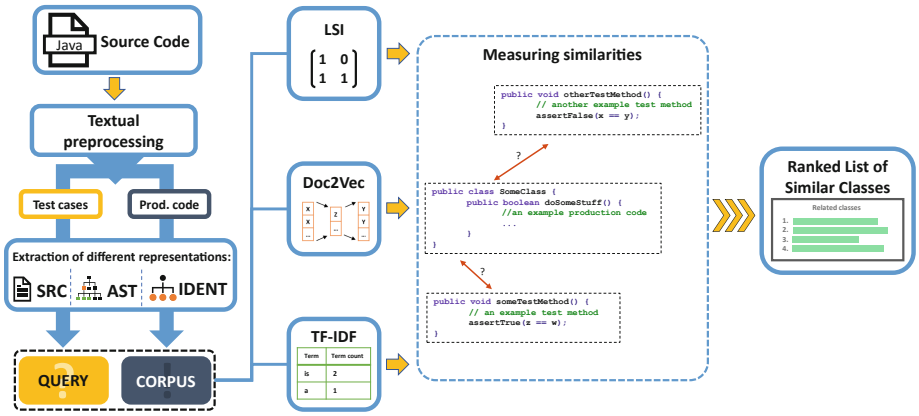


Fig. 1. A high-level illustration of our process.

using the Source Meter[1] static analysis tool and separate test cases and production code. We generate three diverse representations of the source code (SRC, AST, IDENT) which we discuss in Sect. 3 and use machine learning techniques to measure the similarity between code snippets. In the case of Latent Semantic Indexing (LSI) and Term Frequency-Inverse Document Frequency (TF-IDF) methods the models are trained on the production code (corpus) and the test cases are the queries. There is a slight difference in the case of Doc2Vec since the training corpus consists of both the test and production classes. After the models are trained, we measure the similarity between tests and code classes, from which a ranked list is constructed. The basic idea is that test and code classes are *similar* in some sense. Therefore, from the ranked similarity list, we observe the first N production classes, allowing us to consider these techniques as recommendation systems.

We recommend classes for a test case starting from the most similar and also examine the top 2 and top 5 most similar classes. Looking at the outputs in such a way holds a number of benefits. Foremost, if we would consider only the most similar class then those instances when tests assess the proper functioning of several classes rather than only one would be missed. Also, a class usually relies on other classes, consequently a recommendation system can highlight the test and code relationship more thoroughly. Since overly abundant recommendations can result in a high number of false matches which can diminish the usefulness of the information itself, we restricted the consideration to only the 5 most similar classes in each case, keeping the technique as simple as possible. In our current work we used our previously available tools for the generation of LSI-based similarities [5,15].

Summarizing our work, we organize our experiment along three research points, and formulate the following research questions:

[1] https://www.sourcemeter.com/.

RQ1: *How do various source code representations affect the operation of different text-based techniques?*
RQ2: *How the assessed algorithms perform compared to each other?*
RQ2: *Does the combination of these techniques improve traceability link recovery?*

3 Data Collection

We designed our approach to be applicable projects written in Java, one of the most popular programming languages in use [2]. In general, the featured technique is independent of text representations, so the programming language of the source code is not necessarily important. In Fig. 2 one can observe the projects, on which we evaluated our technique. We used the exact same versions of the referenced projects in our previous work [5,15].

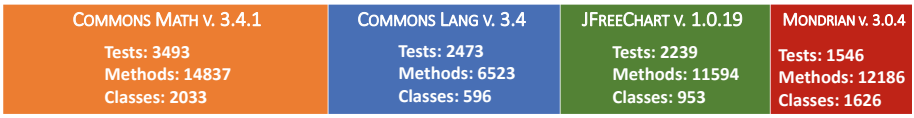


Fig. 2. Size and versions of the examined systems. The area of boxes is proportional to the number of test cases.

The choice of the projects was influenced by several factors: (1) the projects should be publicly available, so we could obtain the source code (2) proper naming convention were followed to some extent (for further information see Sect. 5), and (3) we refer to our previous work, where we used these projects and found they are good representatives for the evaluation. Here we briefly introduce them. Two of the systems strive to have minimal dependencies on other libraries [4] and are modules of the Apache Commons project, these are Commons Lang[2] and Commons Math[3]. Mondrian[4] has a large development history (the development was started in 1997 [26]) and is an open source Online Analytical Processing system, which enables high-performance analysis on massive amounts of data. JFreeChart[5] is a relatively new software, its first release was in 2013. The project is one of the most popular open source charting tools.

From these projects, we obtained three different textual representations to measure similarity. We also described these representations in more detail in our previous work [5]. Similar representations are widely used in other research experiments, such as [38,40]. Here we present brief summary of our representations and through a brief example, we try to better explain them:

[2] https://github.com/apache/commons-lang.
[3] https://github.com/apache/commons-math.
[4] https://github.com/pentaho/mondrian.
[5] https://github.com/jfree/jfreechart.

- **SRC**: In this case, we process the source code as a structured text file. Only splitting [8,12] and stemming are applied. We split the source code along special characters and compound words by the camel case rule. For example, consider the code snippet *int a = 12;* where SRC only splits the text.
- **AST**: Initially, we construct the Abstract Syntax Tree (AST) for a code snippet, then print the type of each node in a pre-order fashion. We employed the publicly available JavaParser[6] tool for AST generation. For the previous example, the representation would be: *VariableDeclarationExpr VariableDeclarator PrimitiveType SimpleName IntegerLiteralExpr*.
- **IDENT**: Like in the previous case, we construct the Abstract Syntax Tree but instead of types we print the only the values and only for the terminal nodes. In the postprocessing phase literals are replaced with placeholders. These printed values are generally the identifiers and constants present in the code. In the previous example, the representation replaces the integer literal, so we get the following sentence: *int a<INT >*.

4 Experiment Design

In this section, we describe the experiments and utilized machine learning methods in detail. Let us consider a straightforward example. In Fig. 3 a simple JUnit test case is presented with its corresponding production code. The code snippet is part of the Commons Lang project. It is easy to see, that the test code chiefly consists of *assert* statements, where the tested methods header is called many times. On the other hand, before the production code, in the comments, the name of the method also often occurs. Notice, that in the IDENT representation the string literals are replaced with placeholders so the method calls will be practically identical in the test case. Although the bodies of the two methods differ, some kind of similarity can be observed. In addition, we measure the similarity between classes, so other methods also contribute to the results. In the upcoming subsections, we explain the techniques used to obtain the similarity between two parts of the source code. We used the Gensim [1] toolkit's implementation for all three machine learning methods.

4.1 Term Frequency–Inverse Document Frequency: TF-IDF

TF-IDF is an information retrieval method, that relies on numerical statistics reflecting how important a word is to a document in a corpus [20]. It is basically a metric and its value increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. One can compute TF-IDF by multiplying a local component (term frequency) with a global component (inverse document frequency) and normalizing the resulting documents to unit length. The formula for a non-normalized weight of term i in document j in a corpus of D documents is displayed in Eq. 1.

[6] https://github.com/javaparser/javaparser.



Fig. 3. An example test case from Commons Lang and the associated production class.

One of the simplest ranking functions is computed by summing the weights for each query term, but many more sophisticated ranking functions also exist [11,28].

$$weight_{ij} = \left(frequency_{ij} * log_2 \frac{D}{DocumentFrequency_i} \right) \tag{1}$$

4.2 Document Embeddings: Doc2Vec

Doc2Vec is originated from Word2Vec, which was introduced by Google's developers in [25]. Word2Vec encodes words into vectors containing real numbers with a neural network, these are called word embeddings. The basic idea is the following: for a given surrounding, the model predicts the current word (CBOW model) or the prediction goes in the opposite direction (Skip-gram model). The trick is that the hidden layer of the shallow neural network used has fewer neurons than the input and output layers, forcing the model to learn a compact representation. The weight in the hidden layers will provide the word embeddings and the number of neurons will be the dimension of the embedding. Doc2Vec differs only in small details: it can encode whole documents by adding a unique identifier of the document to the input layer. This way a word can have multiple embeddings in different documents (which is more realistic in some cases, e.g.: blue, bear). Utilizing the embeddings, we can compute the similarity between documents. We used the 3COSMUL metric proposed in [21], displayed in Eq. 2 to measure similarity between the vectors.

$$arg \max_{b* \in V} \left(\frac{cos(b*, b) cos(b*, a*)}{cos(b*, a) + \epsilon} \right) \tag{2}$$

4.3 Latent Semantic Indexing: LSI

LSI is a technique in natural language processing of analyzing the relationships between documents. During the learning procedure, a matrix is constructed, which contains word counts. The elements inside the matrix are typically weighted with the TF-IDF values, but note that the base process differs from the previous one. The main idea of LSI is that the matrix is transformed into a lower dimension using singular value decomposition and in the resulting matrix the conceptually more similar elements get more similar representations. The most similar documents to a query can easily be found as the query also represents a multidimensional matrix with which a suitable distance method can rank each document by similarity.

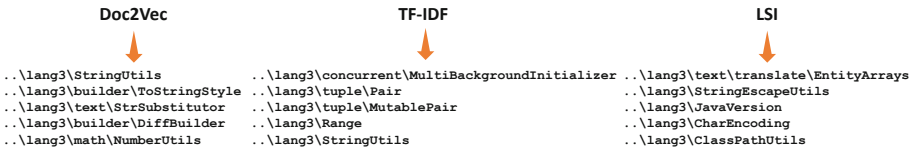


Fig. 4. Ranked lists produced by different approaches for the *StringUtilsSubstringTest* test class.

4.4 Result Refinement with a Combined Technique

After test and code classes had been separated and the code representations had been obtained, we trained the three models separately and investigated the similarities. In Fig. 4 we show the ranked lists of the three alternative methods trained on the IDENT representation. For a given test class (*StringUtilsSubstringTest*) only the Doc2Vec method classified the desired code class (*StringUtils*) as the most similar (while of course in a different case another one of the methods could provide the desired class). Additionally note, that TF-IDF put the *StringUtils* class in the fifth place, while LSI didn't rank it among the top-5 most similar classes (it was in the 11th place of the ranked list). This example demonstrates that the ranked list of each technique can contain useful information, the desired code class appears close to the top of every list. Thus it can be possible to refine the obtained results one technique provides with the list of other techniques. We defined a simple algorithm to achieve this goal, which is shown in Listing 1. We filter the ranked list obtained from the first method with the seconds ranked list. Since the ranked lists contain every code class, we limit them to the top 100 most similar classes, this way the featured algorithm will drop out classes from the first if those are not present on the second ranked list. Note, that this refinement procedure cannot introduce new classes to the first ranked list, only removes them.

```
# ranked_list_i: ranked list from the i-th technique,
 which contains the top 100 most similar classes
result = []
for code_class in ranked_list_1:
    if code_class in ranked_list_2:
        result.append(code_class)
```

Listing 1. Algorithm used to refine the obtained similarity lists.

5 Experiments and Analysis

In this section, we evaluate and discuss the featured text-based models and source code representations. To evaluate our method, we should know whether the proposed machine learning techniques recommend the correct production class for a given test case. To achieve this, we used the existing naming conventions used within the systems and defined the following rules: the class of the test case must possess an identical name as the code class it tests, having the word "Test" before or after the name. Moreover, their directory structure (i.e. package hierarchy) must be the same, so their qualified names also match. For example, if the tested production code class is [CodeClass] than the test should be named [CodeClass*Test*] or [*Test*CodeClass] and their package hierarchy should also match to be considered a correct pair. We calculated precision - the proportion of correctly detected test-code pairs as can be seen in Eq. 3, where the upper part of the fraction denotes how many tests we could retrieve, while the bottom is the number of test cases that match the naming convention. This evaluation strategy is well suited for the listed systems since they are fairly well covered by proper naming conventions.

$$precision = \frac{|relevantTest \cap retrievedTest|}{|retrievedTest|} \tag{3}$$

As detailed in the previous sections, we experimented with three different source code representations and three text-based similarity techniques. We already know, that Doc2Vec and LSI are capable of recovering traceability links, although we do not know how LSI performs on different source code representations. Figure 5 provides a comprehensive collection of the results obtained. From this diagram, it's clear at first glance, that the IDENT representation surpassed all others and AST performed poorly. In some cases, the SRC representation produced quite promising results (e.g.: with TF-IDF at the Commons Math project), but in general, it achieved much lower precision than IDENT. We experienced similar behavior in our previous work, so this confirms our preceding assumption that IDENT is the most suitable representation for test-to-code traceability, since other techniques also performed at lower precision using other representations. Because IDENT seems to be prevalent in finding traceability links correctly, we are going to focus on it in the upcoming discussions.

Answer to RQ1: Based on the results of each evaluated algorithm, we found the IDENT representation to be the most appropriate for finding traceability links correctly.

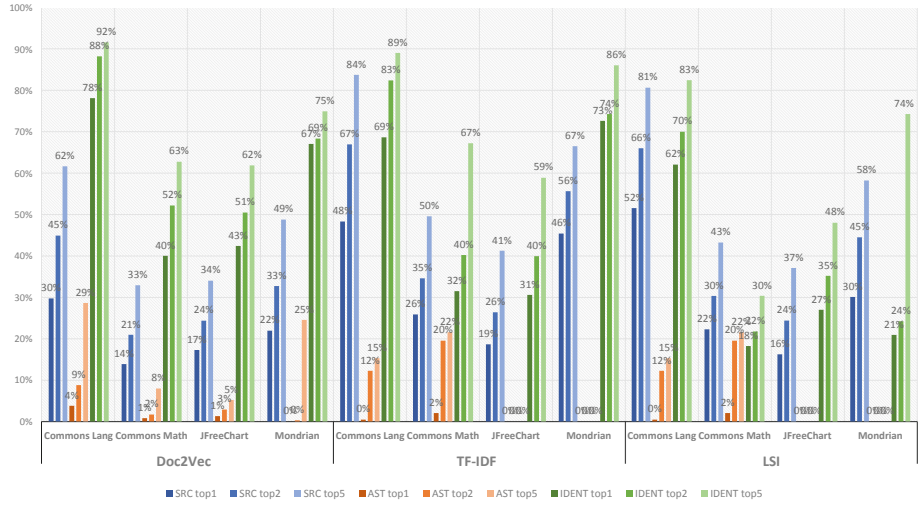


Fig. 5. Results featuring the corpus built from different representations of the source code provided by each technique evaluated.

By examining the results in more detail, we can see, that the precision of Doc2Vec in most cases rises above the rest. Figure 6 presents the results of the featured techniques trained on the IDENT representation. The figure showcases the results we gathered with the most similar classes. It is visible that LSI performed the lowest on the examined systems. TF-IDF's case is a bit unusual since it outperformed the Doc2Vec technique on the Mondrian project with 73% precision rate. The results from other systems also seem quite promising. Therefore from our experiments, Doc2Vec seemed to be the best approach in the test-to-code traceability task, although in some cases other text-based approaches may overperform it. The results of Doc2Vec, in general, are quite convincing, but the obtained matches could still be refined. Because TF-IDF found better results in the Mondrian project than Doc2Vec, combining (for more detail about combination refer to Sect. 5) these methods seems to be a rational idea.

Answer to RQ2: According to the data gathered, we find that in most cases Doc2Vec achieves the best results, although in exceptional cases other text-based techniques can still outperform it.

We investigated the combination possibilities among all the examined techniques. The relation between them is asymmetric, since the first methods ranked list refines the second ones. We found that combining LSI just with TF-IDF only seems to damage the results in both directions, the average precision using this

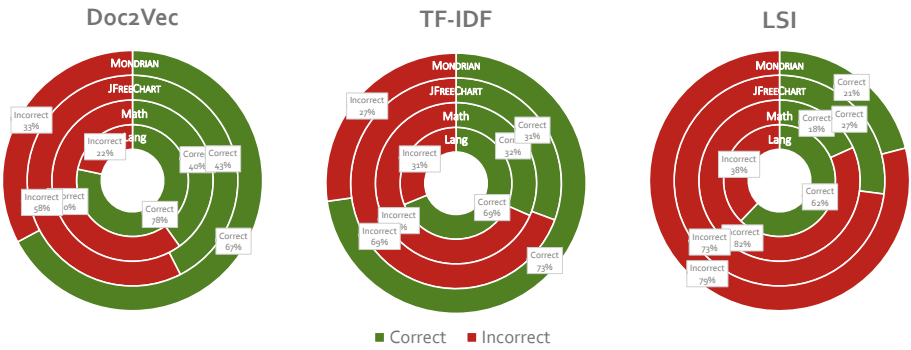


Fig. 6. Results featuring the three approaches, corpus built from the IDENT representation of the source code.

combination was merely 50% for most similar classes, while 70% for to top five items in the ranked list. Combining LSI and TF-IDF with Doc2Vec in such a way that they provide the base of the ranked list and being only refined by Doc2Vec also performed poorly. However, the refinement of Doc2Vec's ranked list with both other techniques improved the results. The refinement of the similarity list with LSI resulted in more than 2.5% improvement compared to Doc2Vec as a standalone technique. Although TF-IDF also improved the values, the improvement was just under 2%. Due to space limitations and to stick to the results that seem more important we do not display these results in detail. We also experimented with the combination of all three techniques: the main similarity list was provided by Doc2Vec and refined by TF-IDF and LSI. When we combined Doc2Vec with both of the other methods, the obtained results were even better than in the other cases. Table 1 presents the results of Doc2Vec as a standalone technique, and as a combination with other approaches. If we compare these results an advantage in the latter method's precision can be recognized. If only the most similar code class is considered, Doc2Vec's average precision was 57%, while with the combined technique we achieved nearly 60%. The most prominent improvement can be identified at Commons Math (roughly 8%), but it is evident that the precision values of every system have increased.

Answer to RQ3: Based on the results, we can clearly see that the combination of techniques has improved the performance in test-to-code traceability link recovery.

While the produced model seems to outperform Doc2Vec in every aspect, Mondrian still remains an outlier. As we saw before, TF-IDF resulted in an outstanding 72.8% precision value, while Doc2Vec merely achieved 67.2%. The deviation is also present if we consider the top 2 or top 5 most similar classes (74%/68% and 86%/75% respectively). While the combined approach improved the outcome of the basic Doc2Vec model, it did not reach the precision of TF-IDF in this single case. Even this considered, on average the combined approach improved the results compared to Doc2Vec by almost 4% (Fig. 7).

Table 1. Results obtained from Doc2Vec as a standalone technique and as a mixed approach.

Top_N results	Doc2Vec			Doc2Vec-TFIDF-LSI		
	1	2	5	1	2	5
Commons Lang	**78.3%**	88.4%	91.8%	**78.3%**	88.4%	93.1%
Commons Math	**40.1%**	52.3%	62.9%	**48.2%**	58.0%	68.2%
JFreeChart	**42.5%**	50.6%	62.0%	**45.3%**	55.2%	67.3%
Mondrian	**67.2%**	68.5%	75.1%	**67.6%**	70.2%	80.7%

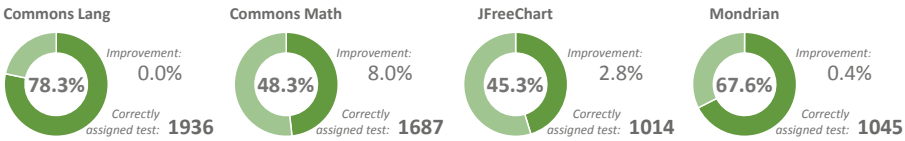


Fig. 7. Result values of the combined Doc2Vec method, trained on the IDENT representation of the source code.

6 Related Work

Traceability in software engineering research typically refers to the discovery of traceability links from requirements or related natural text documentation towards the source code [3,22]. Even as test-to-code traceability is not the most fashionable topic among link recovery tasks, there are several well-known methods that aim to cope with this problem [35]. Test related traceability examples also can be found [5,13,15,31,35], however no known perfect solution exists to the problem. In the research community serious attempts have been made at combating the problem via plugins in the development environment [29] or via static or dynamic analysis [36]. The current state-of-the-art techniques [30] rely on a combination of diverse methods. In this work, we also took advantage of various textual similarity techniques, and the combination of these resulted in a promising recovery precision.

Recommendation systems are also not new to software engineering [18,33,34], presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [5,15].

Word2Vec [25] gained a lot of attention in recent years and became a very popular approach in natural language processing. With this method, calculating similarity between text elements became a mainstream process [9,19,23,27,38,40,43,44]. Textual similarity is useful for example in the clone detection problem [40]. Doc2Vec [24] is an extension of the Word2Vec method dealing with whole documents rather than single words. Although not enjoying the immense popularity of Word2Vec, it is still prominent to the scientific community [6,7,39,46]. In requirement traceability, researchers also made use of word embeddings to recover appropriate links [9,44,45]. Our current approach differs

from these in many aspects. To begin with, we make use of three different similarity concepts, not just one. Next, we compute document embeddings in one step, while in other approaches this is usually achieved in several steps. Finally, our models were trained only on source code (or on some representation which was obtained from the source code) and there was no natural language based corpus.

Using TF-IDF for traceability is not a novelty in the software engineering domain, however, most of the researchers cope with the requirement traceability problem. For example, this technique was used in [42] to trace textual requirement elements to related textual defect reports, or in [10] for the after-the-fact tracing problem. In requirement traceability the use of TF-IDF is so widespread, that it is considered as a baseline method [37]. Our methods did not seem to benefit from TF-IDF as a standalone technique, rather as a refinement for other techniques.

LSI has been applied for recovering traceability links between various software artifacts, even in feature extraction experiments [16]. LSI is already known to be capable of producing good quality results combined with structural information [14,17]. Besides feature extraction, LSI as a standalone technique can be applied to the test-to-code traceability task as well [5,15].

Although natural language based methods are not the most effective standalone techniques, state-of-the-art test-to-code traceability methods like the method provided by Qusef et al. [30,32] incorporate textual analysis for more precise recovery. In these papers, the authors named their method SCOTCH and have proposed several improvements to it. Although their purpose is similar to ours, a fundamental difference is that they used dynamic slicing and focus on the last assert statement inside a test case. Their approach also relies on class name similarity, while we encoded code snippets without any assumptions on naming conventions. These methods use LSI for textual similarity evaluation, while previous evaluations of word embeddings for this purpose are unknown.

7 Conclusions

Test-to-code traceability helps to find production code for a given test case. Our assumption was that the related test and code classes are similar to each other in some sense. We employed three different similarity concepts, based on Doc2Vec, LSI and TF-IDF. Since these methods are intended for natural language texts, we experimented with three different source code representations. Analyzing the obtained data, we derived the conclusion that from simple source code representations, IDENT performs more desirable in test traceability. We compared the obtained results from the three textual similarity techniques and found that the Doc2Vec based similarity performs better in the recovery task than other approaches. Finally, we refined Doc2Vec's ranked similarity list with the recommendation of the other approaches. With this experiment we have successfully improved the performance of Doc2Vec for every project, therefore introducing a successful mixed approach for the textual matching of tests and their production code.

Acknowledgements. This work was supported by the UNKP-18-2 New National Excellence Program and the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). The Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is also acknowledged.

References

1. Gensim: Gensim webpage. https://radimrehurek.com/gensim/. Accessed 2019
2. TIOBE programming community index. https://www.tiobe.com/tiobe-index. Accessed 2019
3. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)
4. Apache Commons webpage (2019). http://commons.apache.org/
5. Csuvik, V., Kicsi, A., Vidács, L.: Source code level word embeddings in aiding semantic test-to-code traceability. In: 10th International Workshop at the 41st International Conference on Software Engineering (ICSE) - SST 2019. IEEE (2019)
6. Dai, A.M., Olah, C., Le, Q.V.: Document Embedding with Paragraph Vectors, July 2015
7. DeFronzo, R.A., et al.: Combination of empagliflozin and linagliptin as second-line therapy in subjects with type 2 diabetes inadequately controlled on metformin. Diab. Care **38**(3), 384–393 (2015)
8. Dit, B., Guerrouj, L., Poshyvanyk, D., Antoniol, G.: Can better identifier splitting techniques help feature location? In: 2011 IEEE 19th International Conference on Program Comprehension (ICPC), ICPC 2011, pp. 11–20. IEEE, Washington, DC (2011)
9. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically enhanced software traceability using deep learning techniques. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017, pp. 3–14. IEEE, May 2017
10. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Improving after-the-fact tracing and mapping: supporting software quality predictions. IEEE Softw. **22**(6), 30–37 (2005)
11. Hiemstra, D.: A probabilistic justification for using tf - idf term weighting in information retrieval. Int. J. Digit. Libr. **3**(2), 131–139 (2000)
12. Hill, E., Binkley, D., Lawrie, D., Pollock, L., Vijay-Shanker, K.: An empirical study of identifier splitting techniques. Empirical Softw. Eng. **19**(6), 1754–1780 (2014)
13. Kaushik, N., Tahvildari, L., Moore, M.: Reconstructing traceability between bugs and test cases: an experimental study. In: 2011 18th Working Conference on Reverse Engineering, pp. 411–414. IEEE, October 2011
14. Kicsi, A., et al.: Feature analysis using information retrieval, community detection and structural analysis methods in product line adoption. J. Syst. Softw. (2019)
15. Kicsi, A., Tóth, L., Vidács, L.: Exploring the benefits of utilizing conceptual information in test-to-code traceability. In: Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, pp. 8–14 (2018)
16. Kicsi, A., Vidács, L., Beszédes, A., Kocsis, F., Kovács, I.: Information retrieval based feature analysis for product line adoption in 4GL systems. In: Proceedins of the 17th International Conference on Computational Science and Its Applications - ICCSA 2017, pp. 1–6. IEEE (2017)

17. Kicsi, A., Vidács, L., Csuvik, V., Horváth, F., Beszédes, Á., Kocsis, F.: Supporting product line adoption by combining syntactic and textual feature extraction. In: Capilla, R., Gallina, B., Cetina, C. (eds.) ICSR 2018. LNCS, vol. 10826, pp. 148–163. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90421-4_10
18. Kochhar, P.S., Xia, X., Lo, D., Li, S.: Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016, pp. 165–176. ACM Press, New York (2016)
19. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. Technical report (2014)
20. Lefebvre-Ulrikson, W., Da Costa, G., Rigutti, L., Blum, I.: Data Mining. New York (2016)
21. Levy, O., Goldberg, Y.: Linguistic regularities in sparse and explicit word representations. Technical report (2014)
22. Marcus, A., Maletic, J.I., Sergeyev, A.: Recovery of traceability links between software documentation and source code. Int. J. Softw. Eng. Knowl. Eng., 811–836 (2005)
23. Mathieu, N., Hamou-Lhadj, A.: Word embeddings for the software engineering domain. In: Proceedings of the 15th International Conference on Mining Software Repositories - MSR 2018, pp. 38–41 (2018)
24. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. Technical report (2013)
25. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems, NIPS 2013, vol. 2, pp. 3111–3119, December 2013
26. Mondrian webpage (2019). http://www.theusrus.de/Mondrian/
27. Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N.: Exploring API embedding for API usages and applications. In: Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017, pp. 438–449. IEEE, May 2017
28. Paik, J.H.: A novel TF-IDF weighting scheme for effective ranking. In: Proceedings of the 36th international ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR 2013, p. 343. ACM Press, New York (2013)
29. Bouillon, P., Krinke, J., Meyer, N., Steimann, F.: EzUnit: a framework for associating failed unit tests with potential programming errors. In: Concas, G., Damiani, E., Scotto, M., Succi, G. (eds.) XP 2007. LNCS, vol. 4536, pp. 101–104. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73101-6_14
30. Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., Binkley, D.: Recovering test-to-code traceability using slicing and textual analysis. J. Syst. Softw. **88**, 147–168 (2014)
31. Qusef, A., Bavota, G., Oliveto, R., De Lucia, A., Binkley, D.: SCOTCH: test-to-code traceability using slicing and conceptual coupling. In: IEEE International Conference on Software Maintenance, ICSM, pp. 63–72. IEEE (2011)
32. Qusef, A., Bavota, G., Oliveto, R., Lucia, A.D., Binkley, D.: Evaluating test-to-code traceability recovery methods through controlled experiments. J. Softw. Evol. Process **25**(11), 1167–1191 (2013)
33. Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Softw. **27**(4), 80–86 (2010)
34. Robillard, M.P., Maalej, W., Walker, R.J., Zimmermann, T. (eds.): Recommendation Systems in Software Engineering. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45135-5

35. Rompaey, B.V., Demeyer, S.: Establishing traceability links between unit test cases and units under test. In: European Conference on Software Maintenance and Reengineering, CSMR, pp. 209–218. IEEE (2009)
36. Sneed, H.: Reverse engineering of test cases for selective regression testing. In: European Conference on Software Maintenance and Reengineering, CSMR 2004, pp. 69–74. IEEE (2004)
37. Sundaram, S.K., Hayes, J.H., Dekhtyar, A.: Baselines in requirements tracing. In: ACM SIGSOFT Software Engineering Notes, vol. 30, p. 1. ACM Press, New York (2005)
38. Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D.: Deep learning similarities from different representations of source code. In: Proceedings of the 15th International Conference on Mining Software Repositories - MSR 2018, vol. 18, pp. 542–553 (2018)
39. Wang, S., Tang, J., Aggarwal, C., Liu, H.: Linked document embedding for classification. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM 2016, pp. 115–124. ACM Press, New York (2016)
40. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016, pp. 87–98 (2016)
41. Wilson, G., et al.: Best practices for scientific computing. PLoS Biol. **12**(1), e1001745 (2014)
42. Yadla, S., Hayes, J.H., Dekhtyar, A.: Tracing requirements to defect reports: an application of information retrieval techniques. Innovations Syst. Softw. Eng. **1**(2), 116–124 (2005)
43. Yang, X., Lo, D., Xia, X., Bao, L., Sun, J.: Combining word embedding with information retrieval to recommend similar bug reports. In: Proceedings - International Symposium on Software Reliability Engineering, ISSRE, pp. 127–137. IEEE, October 2016
44. Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C.: From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering - ICSE 2016, pp. 404–415. ACM Press, New York (2016)
45. Zhao, T., Cao, Q., Sun, Q.: An improved approach to traceability recovery based on word embeddings. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC, vol. 2017-Decem, pp. 81–89. IEEE, December 2018
46. Zhu, Z., Hu, J.: Context aware document embedding, July 2017