Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability

Viktor Csuvik
Department of Software Engineering
University of Szeged
Szeged, Hungary
csuvikv@inf.u-szeged.hu

András Kicsi
Department of Software Engineering
University of Szeged
Szeged, Hungary
akicsi@inf.u-szeged.hu

László Vidács
MTA-SZTE Research Group
on Artificial Intelligence
University of Szeged
Szeged, Hungary
lac@inf.u-szeged.hu

***Abstract*—Proper recovery of test-to-code traceability links from source code could considerably aid software maintenance. Scientific research has already shown that this can be achieved to an extent with a range of techniques relying on various information sources. This includes information retrieval which considers the natural language aspects of the source code. Latent Semantic Indexing (LSI) is widely looked upon as the mainstream technique of this approach. Techniques utilizing word embedding information however also use similar data and nowadays enjoy immense popularity in several fields of study. In this work, we present our evaluation of both LSI and word embeddings in aiding class level test-to-code traceability of 4 open source software systems, the assessment relying on naming convention information.**

***Index Terms*—traceability, testing, test-to-code, word embeddings**

I. Introduction

Software Quality Assurance is a well-researched area, with a serious industrial background. Although quality can be perceived from many viewpoints, testing is one way to gain information about the quality of a software artifact by uncovering faulty parts of the code base. Although testing is considered as a good practice, it cannot identify all the defects of a software, but still, efforts should be made to cover as much production code with tests as possible. Naturally, for larger systems a vast amount of tests is created, tens of thousands of tests are not uncommon in these cases.

The capability to trace software items through a variety of software products is called traceability. In the software engineering domain requirement traceability is the most common problem [1], [2]. In case of a large system when a function or feature has changed, it is often challenging to tell which tests were assessing those part of code. This special task is called *test-to-code traceability*. Although very precise results can be achieved with naming conventions [3] and good coding practices can ease the task, they are no solution for every traceability problem. For instance, if the target system already has numerous tests which lack good coding practices, the task might be practically impossible using only naming conventions. Automatic recovery can still be an option in these cases in the effort of increasing software quality. The topic is already well-known among researchers and several attempts have been made already to cope with this problem [3], [4].

By our perception the current techniques mostly depend on intuitive features thereby making them limited. Our approach maps the code fragments to continuous-valued vectors so that terms used similarly in the source code repository map to similar vectors. We provide a method, that automatically links test cases and production classes relying on only conceptual information.

The contributions of our work are as follows:

- We propose a generic and efficient technique that maps test cases to production classes
- We adapted the doc2vec technique as a traceability link retrieval method, which to the best of our knowledge is a novelty in software engineering
- We show that textual similarities provided by the doc2vec technique approximates the naming convention technique rather well
- We demonstrate that the doc2vec approach can substitute and even outperform the LSI technique in a traceability task

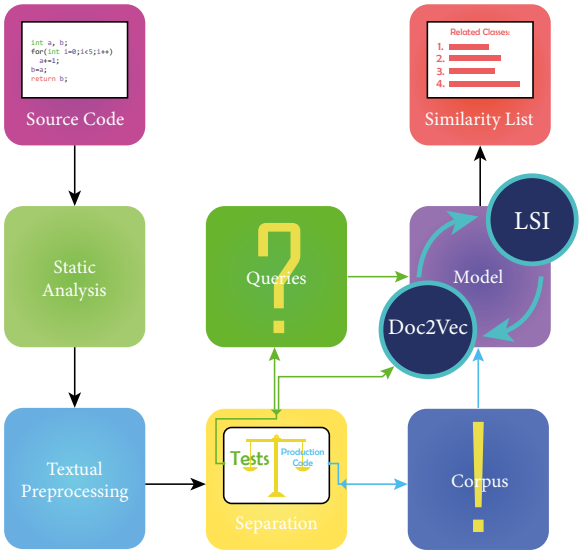

Fig. 1. A high-level illustration of our process

2157-2194/19/$31.00 ©2019 IEEE
DOI 10.1109/SST.2019.00016

II. Research Objectives

In an ideal case the name of the test contains all the needed information. Many developers for example name their test classes after the tested production code as [NameOfTheClassUnderTest*Test*] or [*Test*NameOfTheClassUnderTest]. Thus, the name automatically makes a suggestion to the tested artifact in a structured manner and the intent of it is clear. Other variants of naming conventions can also be found. For more information about good coding practices, please refer to [5].

Let us consider a medium-sized software system with thousands of test cases. As by default all test cases have targets unknown to us, our goal is to find the code class they are meant to test relying on conceptual information. Notice that we have no assumptions about the names of tests, thus the proposed approach is applicable to systems where the naming convention is not followed while writing the test cases. To retrieve traceability links relying only on the source code, we utilize document embedding considering every test case and class as an individual document and computing similarities between them.

We used the doc2vec technique introduced by Mikolov et al. [6]. Word embedding is getting more attention in recent years and was applied in many scientific works. During the search of the ideal learning settings, we tried many approaches proposed in other works. We explain in details these settings in later sections.

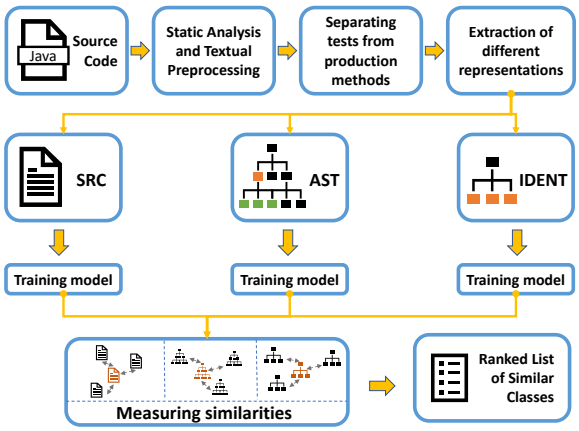

Fig. 2. Illustration of the different doc2vec approaches

Figure 1 shows the comprehensive approach we propose. We work with Java source files from which we extract the required information and separate tests from code functions. We create the LSI model and the diverse doc2vec representations detailed in section III from test cases and production classes. For every representation we train the doc2vec model separately, thus each built training model is different. The obtained vectors contain information about the meaning, environment, and context of a word or document. On these vectors the term similarity is already well-defined, so we can determine class similarities. From the learned similarities we create a ranked list of similar classes for every test case and evaluate these results as in our previous work [4]. We recommend classes for a test case starting from the most similar and also examine the top 2 and top 5 most similar classes. This approach enables our technique to also provide the advantage of a recommendation system in contrast with a strict one-answer based system, thus contributing with more, potentially useful information.

Recovering test-to-code traceability links in a recommendation system manner holds a number of benefits. Ideally, every test case should test only a single class, but in real life applications a unit test can assess the proper functioning of several classes at the same time. A class usually also relies on functionality provided by other classes and cannot function properly without their assistance and correctness. Consequently, a recommendation system can highlight the test and code relationship more thoroughly. On the other hand, too many recommendations could prove frustrating and would not give useful information for the developers. To keep the technique simple and balanced we conducted experiments only with the top 2 and top 5 most similar classes.

To investigate the benefits of the word embedding approach we organize our experiment along the following research questions:

RQ1: *How word embeddings learned on various source code representations perform compared to each other?*

RQ2: *Does external text, namely API documentation, improve link recovery?*

RQ3: *How source code embeddings perform compared to previous text-based approaches?*

III. Methodology

Code representations

For the learning of word embeddings we use 3 different representations of the code: (1) raw source code, (2) type names from abstract syntax tree and (3) identifiers from the abstract syntax tree as visible in Figure 2. Each one requires different steps of pre- and post-processing. Illustrated through a simple example we can see these representations in Figure 3. In this section we describe these representations in detail.

1) Source Code: For a given granularity we consider a source code fragment as a sentence. First we split [7], [8] the text into bag of words representation along opening brackets ("("), opening square brackets ("["), white spaces (" "), punctuations (".") and compound words by the camel case rule. Then we apply stemming to the words. We will refer this representation as *SRC* in the upcoming sections, since in this case we process source code as structured text files.

2) Abstract Syntax Tree: To extract this representation for a code fragment, we initially have to construct an Abstract Syntax Tree (AST). Next, we perform a pre-order visit from the root and for each node we print its corresponding type. Post-processing is not needed on the recovered sentences, every printed node type will be a token in the learning process. We will refer to this representation as *AST* in the upcoming

sections, since it only contains type names from the Abstract Syntax Tree.

3) Identifiers: Like in the previous representation, we extract the Abstract Syntax Tree from the source code. For every node we consider its sub-tree and print the values of the leaf, terminal nodes. Next for a given sentence we replace the constant values with placeholders, corresponding to their types and split words by camel case rule. Next we convert all words to lower case. Similar representations are widely used in other works as well [9], [10]. We will refer to this representation as *IDENT* in the upcoming sections, since we printed out the identifiers (which are substantially the identifiers and constants used in the code) of the Abstract Syntax Tree.

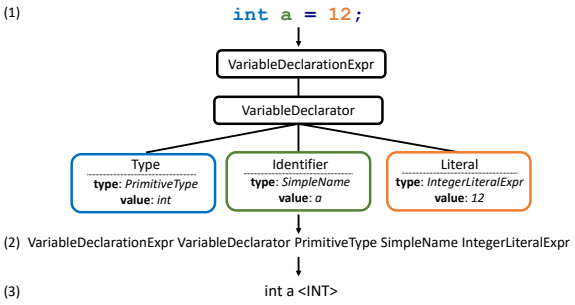

Fig. 3. Different utilized representations of the source code: from the raw source code (1) - *SRC* we construct the abstract syntax tree, then we either print the types of the nodes according to preorder visit (2) - *AST*, or print the values of terminal nodes (3) - *IDENT*.

The publicly available JavaParser[1] was used to generate the representations. It is a lightweight tool made for the analysis of Java code. The AST and IDENT representations were generated separately using only the source code of the examined projects.

Learning Document Embeddings

For a given abstraction level (i.e.: class, method) we select a *document* and extract its representations (SRC, AST and IDENT). We learn document embeddings (an M dimensional vector) for each representation, on which we compute similarities. For every test we detect the most similar production methods. For finding the top-N most similar documents, we used the multiplicative combination objective, proposed in [11], displayed in Equation 1. Here V represents the vocabulary, a, $a*$ and b, $b*$ are two pairs of words that share a relation in a word-analogy task, while ϵ is used to prevent divisions by zero. Since documents consist of words (and doc2vec relies greatly on word2vec) this metric relies mostly on word embeddings. This means that positive words still contribute positively towards the similarity, negative words negatively, but with less susceptibility to one large distance dominating the calculation. The metric is applicable to document level, because the doc2vec model is very similar to word2vec: instead of using just surrounding words to predict the next word, we also add another feature vector, which is unique for every sentence. This way a single word can have different embeddings in different sentences. Thus, the formula below can be used similarly to word2vec, the roles of the vocabulary and the selected words are also identical.

$$arg\max_{b* \in V} \left(\frac{cos(b*, b) cos(b*, a*)}{cos(b*, a) + \epsilon} \right) \quad (1)$$

Doc2vec is basically a fully connected neural network, which uses a single hidden layer to learn document embeddings. We feed the input documents to this neural network for each representation and it computes conceptual similarity of these documents. In our current experiments we are not interested in every similarity, just the ones between test cases and production methods.

IV. Data Collection

In this paper we worked with projects written in the Java programming language. Since word embeddings are independent from languages (and in general from any kind of representations), the programming language of the source code is not necesarrily important. In theory, only difference should be in the way how AST representations are produced and the type of the nodes. In Table I we listed the systems on which we evaluated our approach. These are the exact same versions of the referenced projects as in our previous paper [4] on LSI-based test-to-code traceability.

Commons Lang[2] and Commons Math[3] are both modules of the Apache Commons project. Lang provides helper utilities for the java.lang API, while Math is a library which contains mathematical and statistical components for problems not available in Java programming language. JFreeChart[4] is an open source library developed in Java and currently is one of the most widely used charting tool among developers. Mondrian[5] is an open source Online Analytical Processing (OLAP) server system. It enables high performance analysis on large amount of data.

The main motivation behind project selection was that for these projects we could check whether our approach connected the test cases with the appropriate production classes and to compare these with our previous results. For further discussion about evaluation see Section V. The selected systems are also publicly available, obtaining the source code is not a problem. Commons Lang and Commons Math are widely used and strive to have minimal dependencies on other libraries [12]. Starting in 1997 [13], Mondrian has a large history and significantly less developers were involved in it compared to Apache Commons projects. In contrast, JFreeChart's first release was in 2013 [14], this makes it a rather new development. Although the shown projects are diverse, we cannot state, that they represent every characteristics of a general Java system. The proposed approach can be applied to every Java system where test cases are present but our

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

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

current evaluation can work only with systems where naming conventions were followed.

TABLE I
SIZE AND VERSIONS OF THE SYSTEMS EXAMINED

Program	Version	Classes	Methods	Tests
Commons Lang	3.4	596	6523	2473
Commons Math	3.4.1	2033	14837	3493
JFreeChart	1.0.19	953	11594	2239
Mondrian	3.0.4.11371	1626	12186	1546

In our approach, we first separate methods from the source code. We extract the text of the methods with the help of the Source Meter [15] static analysis tool. We classify these to test methods and production methods. After this, we produce the representations specified in section III. To construct the Abstract Syntax Tree for every source file we used the JavaParser [16] package. Since the goal of doc2vec is to create a numeric representation of any text regardless of its length, we have to define what a *document* is in our case. In the current experiments, we considered every class to be a unique document. This way we produced class level similarities between test and code classes to determine the conceptual connections between them. In the experiments we chose the Gensim [17] toolkit's implementation of doc2vec. The doc2vec model has many hyperparameters, conceivably the most important amongst them is the embedding size [18] (the number of neurons in the hidden layer). We experimented with multiple size settings and found that we obtained the most promising results with vectors of 100 dimensions. Window size (gram window size - number of words to examine to left and to the right) and number of epochs (number of steps for training the model) are also essential, we set these values to 10 and 20 respectively. We worked with the distributed bag of words (PV-DBOW) training algorithm. During the training it can also be specified that if words appear less than a given number will be ignored. We adjusted this setting to 2, though it does not play a large role since in source codes a limited set of tokens are used - it's especially true for Abstract Syntax Tree types. The model features a vocabulary, which keeps track of all unique words, sorts words by frequency and after training it can be queried to use the obtained embeddings in various ways. Both initial learning rate and ending learning rate were left according to the original settings of Gensim.

A. Training Set Experiments

Besides examining a diverse set of representations we also experimented with different training corpora, on which the doc2vec model was trained. Accordingly, we carried out three different configurations: (1) a corpus built from every system separately for individual learning, (2) a corpus built from a common code base and (3) a corpus supplement with API documentation. In this section, we describe these experiments in details.

1) Programs Trained Separately: This is the most classic and straightforward scenario. For a given project we train the model on its own code base for each representation. The size of the training set depends on the project and it exclusively contains relevant information from the point of the system. Siwei et al. [19] state that when training word embeddings, the corpus domain is more important than the corpus size, and using an in-domain corpus significantly improves the performance. To better understand this statement, we defined the TC metric, which denotes in general how many tests are written for a single class: TC = Number of Test Cases / Number of Classes. In figure 4 one can observe, how the TC metric moves along with the computed doc2vec results. Although we cannot draw a conclusion from the measured values, it seems that from a bigger corpus training the model gives a more general result, and also has a chance to learn cross-system similarities, yet this is not a rule without exception as we can see in the case of Mondrian. Table II shows the results from this experiment.

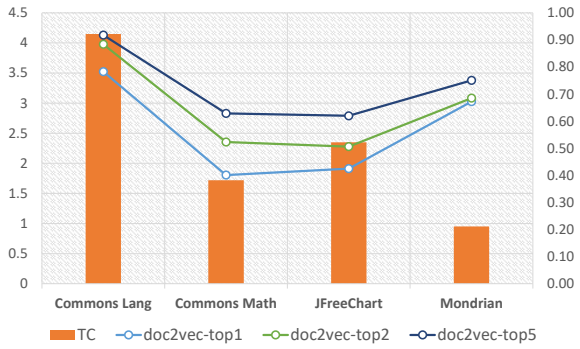

Fig. 4. Correlation between TC = Number of Test Cases / Number of Classes and results from our approach

2) Training on Joint Corpus: In this case, we created a joint training set from the projects listed. The size of the corpus was 1.72 GB for AST representation, 25 MB for SRC and 770 MB for IDENT. The training time of the model also increased a lot. The fundamental challenge was that for a given code fragment (e.g.: class, method) its most similar pair could be in other systems. To avoid references across different systems, we filtered out these results. This way every system gets matches from their own code base. To identify cross-references for a test case we simply deleted classes of other systems from the tests ranked list of similar classes. Without eliminating these references we would get messy results and the meaning of these links is also unclear. Similar filtering was applied to test cases, because for a given test doc2vec gives us back the most similar embeddings (considering every sentence in the train corpus). Amongst these sentences the first was typically the examined test case (every test is the most similar to itself), and other test cases could also appear which should be removed. Without eliminating these cross-references the models performance would drop significantly, since the evaluation only considers internal connections. Although it is

conceivable to have a test case that connects to other system, or the same test is present in multiple systems, but these examinations are out of scope for this paper.

3) Training on API Documentation: The training configurations above trained the model only on a given code representation (SRC, AST, IDENT). It means, that the model learned natural language relationships and structure only from some identifiers and comment snippets. Software documentation could also provide useful semantic information about a system [20]. Also, the corpus on which we train the model is extremely important [21]. To examine this idea, we picked three projects that have standard JavaDoc documentation to train on: Commons Lang, Commons Math, and JFreeChart. Unfortunately, we did not find a standard JavaDoc page for Mondrian from where we could mine contextual data automatically. We trained the model on the three representations (discussed at length in III), then we supplemented the corpus in two ways: (1) with a brief description of the methods (2) with the previous sentences and with the long package description. These information can be found at the projects' JavaDoc website. The brief descriptions are listed next to each method in an HTML table, while the package description is at the bottom of the page. Text information was automatically extracted from publicly accessible JavaDoc documentation websites using a python script. Even as the two differ in fundamental ways, we did not find big differences in the results. Still, the latter technique still seems to suit our purpose better, so we display these in Table III.

V. Results and Discussion

In this section, we evaluate the proposed technique and publish the results. The reader can study the results we obtained from different representation and learning settings. In addition, we compare the doc2vec approach with our previous work [4], where we applied Latent Semantic Indexing (LSI) for the same task. Table IV shows the outcomes of LSI, while Table II and Table III show results obtained with via embeddings. The results are summarized in Figure 5 for the most similar classes.

According to Rompaey et al. [3] total precision can be achieved in the test-to-code traceability task using proper name conventions. However appropriate naming is often imperfect and thus the employment of contextual recommendation systems can still be relevant. The systems we listed above are fairly well covered by proper naming conventions. To determine test-code pairs, we defined the following simple algorithm: the class of the test case must have the same name as the code class it tests, having the word "Test" before or after the name. In addition, their package hierarchy must be the same, thus their qualified names are also matching. To evaluate our procedure, we compared our results to the pairs obtained through naming conventions. We calculated precision - the proportion of correctly detected test-code pairs as can be seen in Equation 2 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.

$$precision = \frac{|relevantTest \cap retrievedTest|}{|retrievedTest|} \quad (2)$$

RQ1: Word embeddings learned on various source code representations

As detailed in previous sections, we experimented with three source code representations and three training configurations. In this section we evaluate them. Table II shows the results from the featured representation, where every project was trained separately. It can be read from the table that the IDENT method supplied the best results by far. We found that in the joined corpus training scenario results were irredeemably low, we found them not to be worth highlighting. The AST and SRC representations also resulted in much lower precision, we did not find them worthy of further examination. Even if these attempts have not proved to be overly successful in recovering test-to-code links, for other tasks, for example code clone detection they may still be applicable.

Answer to RQ1: The IDENT representation combined with separately trained projects seems to be prevalent in finding traceability links correctly, while other methods and scenarios proved to be much less effective.

RQ2: Using API documentation

API documentation is a software artifact containing instructions and information about the inspected library. It is basically a natural text, supplemented with technical details about the software. The goal of this experiment is to get more insight on whether including API documentation to the embedding learning process improves traceability performance. We examined every representation we featured above but in this section, we feature only the IDENT method, which performed best in RQ1. While doc2vec showed improvement compared to LSI, the corpus expanded with API documentation did not increase by a lot. In Table III we can see, that the numbers are almost the same compared to the original measurements. Moreover, the first 2 and 5 elements of the ranked list seem to be less useful than before. The results at Commons Lang and Commons Math improved by nearly 2%, at JFreeChart it dropped by 5%, which is quite notable.

Answer to RQ2: Judging by the numbers, the inclusion of API documentation produces similar results to the base scenario but behaves unpredictably at some systems. Its use can be subject to further examination but a high change in precision is not likely.

RQ3: Source code embeddings compared to LSI-based traceability

It is known that LSI is capable of recovering traceability links. In this section we compare our baseline data from [4] with the doc2vec results. The two datasets are shown in Table IV. The LSI technique gives an average of 35% precision

TABLE II
RESULTS FEATURING THE CORPUS BUILT FROM DIFFERENT REPRESENTATIONS OF THE SOURCE CODE, SYSTEMS TRAINED SEPARATELY

Program	SRC			AST			IDENT		
	$doc2vec_{top1}$	$doc2vec_{top2}$	$doc2vec_{top5}$	$doc2vec_{top1}$	$doc2vec_{top2}$	$doc2vec_{top5}$	$doc2vec_{top1}$	$doc2vec_{top2}$	$doc2vec_{top5}$
Commons Lang	**29.8%**	45.0%	61.8%	**3.8%**	8.8%	28.7%	**78.3%**	88.4%	91.8%
Commons Math	**13.9%**	21.0%	33.0%	**0.8%**	1.7%	8.0%	**40.1%**	52.3%	62.9%
JFreeChart	**17.3%**	24.4%	34.1%	**1.3%**	2.9%	5.2%	**42.5%**	50.6%	62.0%
Mondrian	**22.0%**	32.8%	48.9%	**0.0%**	0.3%	24.6%	**67.2%**	68.5%	75.1%

TABLE III
RESULTS FEATURING THE CORPUS BUILT FROM THE IDENT REPRESENTATION OF THE SOURCE CODE AND API DOCUMENTATION

Program	$doc2vec_{top1}$	$doc2vec_{top2}$	$doc2vec_{top5}$
Commons Lang	**78.8%**	85.9%	93.1%
Commons Math	**43.9%**	51.9%	65.7%
JFreeChart	**42.1%**	48.4%	57.7%

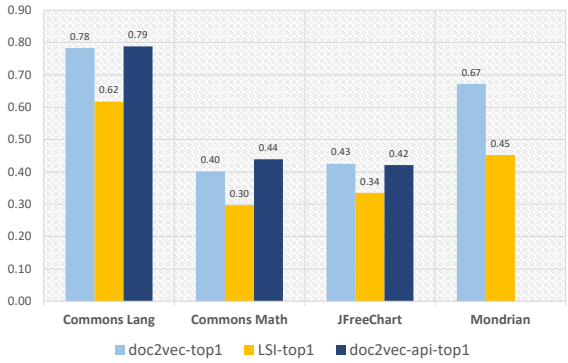

Fig. 5. Results featuring doc2vec and LSI techniques in text-to-code recovering precision

in these systems, and the top 5 elements in the ranked list increase this result by approximately 30%. Beside the LSI results, we can view the precision provided by doc2vec. If we compare these results we can see a significant advantage in the latter method's precision. The average precision of our approach is nearly 60%, which almost doubles the results of LSI. Although these results are quite promising, the increase varies between systems: Commons Math gained the least with about 10%, while the most increase appeared at Mondrian with a remarkable 22% enhancement. As the quality of the output significantly improved for the first match, the top 2 and top 5 results also gained more precision. The average improvement for the top 2 is approximately 10%, while in the top 5 is less significant. It is clear by our evaluation that doc2vec as a standalone technique outperforms LSI, while as a recommendation system they perform similarly. It is also evident that there are serious differences between projects. We believe that the developer habits and even the size of the projects influence the results.

Answer to RQ3: Based on our data we conclude that document embeddings can substitute other current text-based approaches in test-to-code traceability methods and with correct configuration even outperform them greatly.

Figure 5 shows the relationship between different measurements in a more comprehensible perspective. The figure showcases the results we got with the most similar classes. It is evident that the precision values of every system have increased.

A single result doc2vec is often not perfect, however, as a recommendation system it gives a 73% precision on average. Also, it produces results even where naming conventions were not applied, although test cases that do not follow naming conventions may also impair the results in this scenario.

VI. RELATED WORK

Traceability in software engineering is a rather well-researched topic, relevant research is mostly in the direction of requirements traceability or traceability of natural text documents [1], [2] and there are even examples of test related traceability initiatives [3], [4], [22], [23]. There are several well-known methods [3] for test-to-code traceability also and serious attempt has been made at combating the problem via plugins in the development environment [24] or via static or dynamic analysis [25]. The current state-of-the-art techniques [26] rely on a combination of different methods.

Recommendation systems are also not new to software engineering [27]–[29], presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [4].

During the recent years of natural language processing, word2vec [30] has become a very popular approach of calculating semantic similarities between various textual holders of information [9], [10], [21], [31]–[35]. Our current approach differs from these in many aspects. First, we compute document embeddings in one step, while in most cases initially word embeddings are obtained and then the authors propagate these to larger text body (e.g.: sentences, documents). Next, there is no natural language corpus from the model can learn contextual interrelations, only source code files. This makes the learning phase more difficult and requires a good representation of source files and an appropriate configuration of the model.

Related research also focuses on artifacts written in natural language and use NLP techniques for various purposes

TABLE IV
RESULT VALUES OF OUR PREVIOUS WORK COMPARED TO OUR CURRENT EXPERIMENTS USING IDENT REPRESENTATION OF SOURCE CODE

Program	LSI			doc2vec		
	LSI_{top1}	LSI_{top2}	LSI_{top5}	$doc2vec_{top1}$	$doc2vec_{top2}$	$doc2vec_{top5}$
Commons Lang	**61.7%**	73.6%	88.6%	**78.3%**	88.4%	91.8%
Commons Math	**29.7%**	42.3%	56.9%	**40.1%**	52.3%	62.9%
JFreeChart	**33.5%**	45.7%	62.9%	**42.5%**	50.6%	62.0%
Mondrian	**45.2%**	58.4%	73.1%	**67.2%**	68.5%	75.1%

and some even employ word embeddings. Int traceability requirements traceability is usually the most popular issue. Jin Guo et al. [33] proposed a network structure, where firstly word embeddings are learned and then a recurrent neural network uses these vectors to learn the sentence semantics of requirements artifacts. Their main perceived problem seemed to be that the architecture was only trained to process natural language text and could not handle other types of artifacts like source code or AST representation. Zhao et al [36] present an approach which is based on word embeddings to rank the recovered traceability links. In this work, an approach named WELR was presented, which is based on word embeddings to rank the recovered traceability links in bug localization. Their method consists of two phases: in the first phase input text data is being preprocessed and the model is trained, while in the second phase they rank the traceability links. In [32], the authors propose an architecture where word embeddings are trained on API documents, tutorials and reference documents, and then aggregated in order to estimate semantic similarities between documents which they used for bug localization purposes. Their setting was similar to ours, although they did not take advantage of the AST representation. Word embeddings can, in addition, be used to find similarity between sentences/paragraphs/documents [10]. In the software engineering domain, this can also be useful in clone detection.

Doc2vec [6] 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 well-known to the scientific community [37]–[40], although they are much less prevalent in the field of software engineering. To the best of our knowledge, we are the first who use doc2vec to recover traceability links.

Although natural language based methods are not the best standalone techniques, state of the art test-to-code traceability methods like the method provided by Qusef et al. [26], [41] 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. However, those methods use LSI for textual similarity evaluation, while previous evaluation of word embedding for this purpose is not known.

VII. CONCLUSIONS

Natural language processing methods are widely applied in software engineering research, including traceability link recovery. In this paper, we employ word embedding methods on source code to find test-to-code traceability links. Since these methods are intended for natural language texts, we first experimented with various representations of source code. We found that AST-based identifier extraction is the most appropriate way to learn source code embeddings. The second idea was to enrich this representation with natural language text from API documentation, but its usefulness is not evident from the data we measured. Third, we compared the source code embeddings with LSI based similarity and found that with appropriate representation embeddings perform better, thus presenting a valuable alternative in test-to-code traceability research.

ACKNOWLEDGEMENT

This work was supported in part by 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] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, oct 2002.

[2] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of Traceability Links between Software Documentation and Source Code." *International Journal of Software Engineering and Knowledge Engineering*, pp. 811–836, 2005.

[3] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *European Conference on Software Maintenance and Reengineering, CSMR*. IEEE, 2009, pp. 209–218.

[4] A. Kicsi, L. Tóth, and L. Vidács, "Exploring the benefits of utilizing conceptual information in test-to-code traceability," *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pp. 8–14, 2018.

[5] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson, "Best Practices for Scientific Computing," *PLoS Biology*, vol. 12, no. 1, p. e1001745, jan 2014.

[6] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," Tech. Rep., 2013.

[7] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, dec 2014.

[8] "Can better identifier splitting techniques help feature location?" in *IEEE International Conference on Program Comprehension*. IEEE, jun 2011, pp. 11–20.

[9] "Deep learning similarities from different representations of source code," *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, vol. 18, pp. 542–553, 2018.

[10] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pp. 87–98, 2016.

[11] "Linguistic Regularities in Sparse and Explicit Word Representations," Tech. Rep., 2014.

[12] "Apache Commons webpage," http://commons.apache.org/, 2019.

[13] "Mondrian webpage," http://www.theusrus.de/Mondrian/, 2019.

[14] "JFreeChart webpage," http://www.jfree.org/jfreechart/, 2019.

[15] "SourceMeter sourcemeter webpage," https://www.sourcemeter.com/, accessed: 2019.

[16] "JavaParser javaparser webpage," http://javaparser.org, accessed: 2019.

[17] "Gensim gensim webpage," https://radimrehurek.com/gensim/, accessed: 2019.

[18] W. Fu and T. Menzies, "Easy over Hard: A Case Study on Deep Learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. New York, New York, USA: ACM Press, 2017, pp. 49–60.

[19] S. Lai, K. Liu, S. He, and J. Zhao, "How to generate a good word embedding," *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 5–14, nov 2016.

[20] "Automated construction of a software-specific word similarity database," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings*. IEEE, feb 2014, pp. 44–53.

[21] N. Mathieu and A. Hamou-Lhadj, "Word embeddings for the software engineering domain," *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pp. 38–41, 2018.

[22] N. Kaushik, L. Tahvildari, and M. Moore, "Reconstructing Traceability between Bugs and Test Cases: An Experimental Study," in *2011 18th Working Conference on Reverse Engineering*. IEEE, oct 2011, pp. 411–414.

[23] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "SCOTCH: Test-to-code traceability using slicing and conceptual coupling," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 2011, pp. 63–72.

[24] F. S. Philipp Bouillon, Jens Krinke, Nils Meyer, "EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors," in *Agile Processes in Software Engineering and Extreme Programming*. Springer Berlin Heidelberg, 2007, vol. 4536, pp. 101–104.

[25] H. Sneed, "Reverse engineering of test cases for selective regression testing," in *European Conference on Software Maintenance and Reengineering, CSMR 2004*. IEEE, 2004, pp. 69–74.

[26] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147–168, 2014.

[27] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[28] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, jul 2010.

[29] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems*, vol. 2, pp. 3111–3119, dec 2013.

[31] Q. V. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," Tech. Rep., 2014.

[32] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 404–415.

[33] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically Enhanced Software Traceability Using Deep Learning Techniques," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*. IEEE, may 2017, pp. 3–14.

[34] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. IEEE, oct 2016, pp. 127–137.

[35] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*. IEEE, may 2017, pp. 438–449.

[36] T. Zhao, Q. Cao, and Q. Sun, "An Improved Approach to Traceability Recovery Based on Word Embeddings," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2017-Decem. IEEE, dec 2018, pp. 81–89.

[37] Z. Zhu and J. Hu, "Context Aware Document Embedding," jul 2017.

[38] A. M. Dai, C. Olah, and Q. V. Le, "Document Embedding with Paragraph Vectors," jul 2015.

[39] S. Wang, J. Tang, C. Aggarwal, and H. Liu, "Linked Document Embedding for Classification," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16*. New York, New York, USA: ACM Press, 2016, pp. 115–124.

[40] R. A. DeFronzo, A. Lewin, S. Patel, D. Liu, R. Kaste, H. J. Woerle, and U. C. Broedl, "Combination of empagliflozin and linagliptin as second-line therapy in subjects with type 2 diabetes inadequately controlled on metformin," *Diabetes Care*, vol. 38, no. 3, pp. 384–393, jul 2015.

[41] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Evaluating test-to-code traceability recovery methods through controlled experiments," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1167–1191, nov 2013.