IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 20, 2021, accepted May 4, 2021, date of publication May 26, 2021, date of current version June 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3083923

Large Scale Evaluation of Natural Language Processing Based Test-to-Code Traceability Approaches

ANDRÁS KICSI, VIKTOR CSUVIK, AND LÁSZLÓ VIDÁCS
MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, 6720 Szeged, Hungary
Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary

Corresponding authors: András Kicsi (akicsi@inf.u-szeged.hu) and László Vidács (lac@inf.u-szeged.hu)

This work was supported in part by the New National Excellence Programs under Grant ÚNKP-20-3-SZTE and Grant ÚNKP-20-5-SZTE, in part by the Ministry of Innovation and Technology, Hungary, under Grant NKFIH-1279-2/2020N, and in part by the Artificial Intelligence National Laboratory Programme of the National Research, Development and Innovation (NRDI) Office of the Ministry of Innovation and Technology. The work of László Vidács was supported by the János Bolyai Scholarship of the Hungarian Academy of Sciences.

ABSTRACT Traceability information can be crucial for software maintenance, testing, automatic program repair, and various other software engineering tasks. Customarily, a vast amount of test code is created for systems to maintain and improve software quality. Today's test systems may contain tens of thousands of tests. Finding the parts of code tested by each test case is usually a difficult and time-consuming task without the help of the authors of the tests or at least clear naming conventions. Recent test-to-code traceability research has employed various approaches but textual methods as standalone techniques were investigated only marginally. The naming convention approach is a well-regarded method among developers. Besides their often only voluntary use, however, one of its main weaknesses is that it can only identify one-to-one links. With the use of more versatile text-based methods, candidates could be ranked by similarity, thus producing a number of possible connections. Textual methods also have their disadvantages, even machine learning techniques can only provide semantically connected links from the text itself, these can be refined with the incorporation of structural information. In this paper, we investigate the applicability of three text-based methods both as a standalone traceability link recovery technique and regarding their combination possibilities with each other and with naming conventions. The paper presents an extensive evaluation of these techniques using several source code representations and meta-parameter settings on eight real, medium-sized software systems with a combined size of over 1.25 million lines of code. Our results suggest that with suitable settings, text-based approaches can be used for test-to-code traceability purposes, even where naming conventions were not followed.

INDEX TERMS Software testing, unit testing, test-to-code traceability, natural language processing, word embedding, latent semantic indexing.

I. INTRODUCTION

The creation of quality software usually involves a great effort on part of developers and quality assurance specialists. The detection of various faults is usually achieved via rigorous testing. In a larger system, even the maintenance of tests can be a rather resource-intensive endeavor. It is not exceptional for software systems to contain tens of thousands of test cases each serving a different purpose. While their aims can be self-evident for their authors at the time of their creation, they bear no formal indicator of what they are meant to test. This can encumber the maintenance process. The problem of locating the parts of code a test was meant to assess is commonly known as test-to-code traceability.

Proper Test-to-Code traceability would facilitate the process of software maintenance. Knowing what a test is supposed to test is obviously crucial. For each failed test case, the code has to be modified in some way, or there is little point to testing. As this has to include the identification of the production code under test, finding correct test-to-code traceability links is an everyday task, automatization would be beneficial. This could also open new doors for

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

fault localization [1], which is already an extensive field of research, and even for automatic program repair [2], greatly contributing to automatic fixes of the faults in the production code.

To the best of our knowledge, there is no perfect solution for recovering the correct traceability links for every single scenario. Good testing practice suggests that certain naming conventions should be upheld during the testing, and one test case should strictly assess only one element of the code. These guidelines, however, are not always followed, and even systems that normally strive to uphold them contain certain exceptions. Thus, the reliability of recovery methods that build on these habits can differ in each case. Nonetheless, the method of considering naming conventions is one of the easiest and most precise ways to gather the correct links.

In its simplest form, maintaining naming conventions means that the name of the test case should mirror the name of the production code element it was meant to test, its name consisting of the name of the class or method under test and the word ''test'' for instance. The test should also share the package hierarchy of its target. In a 2009 work of Rompaey and Demeyer [3] the authors found that naming conventions applied during the development can lead to the detection of traceability links with complete precision. These, however, are rather hard to enforce and depend mainly on developer habits. Additionally, method-level conventions have various other complicating circumstances.

Other possible recovery techniques rely on structural or semantic information in the code that is not as highly dependent on individual working practice. One such technique is based on information retrieval (IR). This approach relies mainly on textual information extracted from the source code of the system. Based on the source code, other, not strictly textual information can also be obtained, such as Abstract Syntax Trees (AST) or other structural descriptors. Although source code syntax is rather formal and most of the keywords of the languages are given, the code still usually contains a large amount of unregulated natural text, such as variable names and comments. There are endless possibilities in the naming of variables, functions, and classes. These names are usually quite meaningful. While source code is hard to interpret for humans as natural language text, machine learning (ML) methods commonly used in natural language processing (NLP) could still function properly.

Compared to a small manual dataset, Rompaey and Demeyer [3] found that lexical analysis (Latent Semantic Indexing - LSI) applied to this task performed with 3.7%-13% precision while the other methods all achieved better results. Thus, it is known that IR-based methods most probably do not produce the best results in the test-to-code traceability field. However, they are in constant use in current state-of-the-art solutions. Textual methods may not be the single best way to produce valid traceability links, but modern approaches still employ them in combination with other techniques. The textual methods used in these systems are usually less current, most solutions simply rely on matching class names or the latent semantic indexing (LSI) technique as part of their contextual coupling. Thus, finding better performing textual methods can improve these possible combinations as well, having the potential of major contributions to the field. Our findings [4] show that improved versions of lexical analysis can significantly outshine the previously mentioned low results, raising their average precision over 50%.

To investigate the benefits of ML models and to point out their distinction from simple naming conventions, our experiments are organized along the following research questions:

- **RQ1:** How generally are naming conventions applied in real systems?
- **RQ2:** Is there a way to further improve test-to-code traceability results relying on modern information retrieval methods?
- **RQ3:** How well do various text-based techniques perform compared to human data?

In the current paper, our goal is to recover test-to-code traceability links for tests based on only the source files. To do so, a suitable input representation is generated and from this, an artificial intelligence model is trained for the search of the most similar test-to-code match.

The paper is organized as follows. Section II presents diverse background information, including our various approaches to input generation and traceability link recovery. Our evaluation procedure and the sample projects are also described in this section. An evaluation on eight systems follows in Section III with the discussion of these results in Section IV. Related work is overviewed in Section V, some threats to validity are addressed in Section VI, while our paper concludes in Section VII.

II. THE PROPOSED METHOD

The goal of the current paper is to investigate textual techniques for the sake of improving test-to-code traceability. This approach could improve the performance of existing techniques on this specific problem and also serve as the groundwork for future works on test-to-code traceability. To achieve this, let us grasp the process in Figure 1. The input is a software system, which consists of Java source files. The output is a ranked list for each test case with the production classes that are likely to be a target of the test. The input files are transformed in such a representation, that is more suited to machine learning than raw source code. Three techniques are trained to measure the similarity between test and code classes. Class information is also obtained from the source files like the list of imported packages and the methods defined in a class. Each model produces a list of similar code classes but these results are susceptible to faults because of the nature of ML techniques. Thus, these lists are filtered with the class information which was obtained earlier.

Our research strives to achieve a comprehensive evaluation of three text-based techniques on the test-to-code traceability problem rather than simply providing a new method. Thus, our results were evaluated on eight real open-source programs

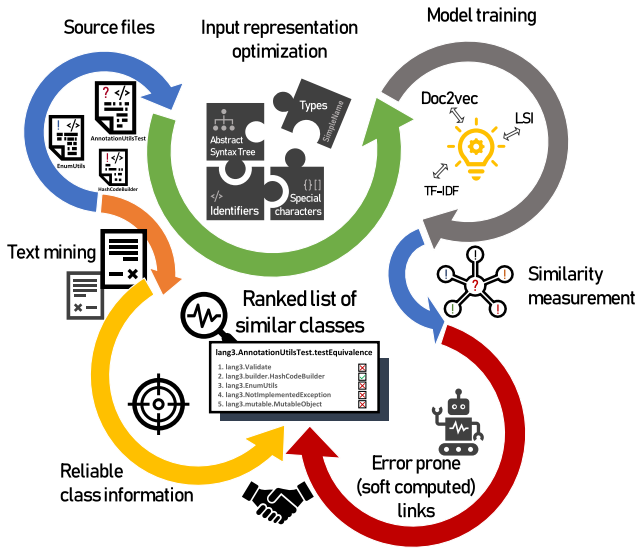

FIGURE 1. A high level overview of the proposed process.

and also using a variety of source code representations and settings. Our previous work aimed to show that LSI itself performs better than it was previously perceived by the research community [5], investigated the question of source code representations in the task, and also found that Doc2Vec can significantly outperform LSI [6] while a suitable combination of the textual similarity techniques could provide even better results [4]. Some of the approaches used in the paper are also defined in our previous work but they are also briefly introduced in the following subsections.

A. LATENT SEMANTIC INDEXING

LSI is not a relatively old algorithm and there is also previous work on its uses on this specific problem. It builds a corpus from a set of documents and computes the conceptual similarity of these documents with each query presented to it. In our current experiments, the production code classes of a system were considered as the documents forming the corpus, while the test cases were considered as queries. The algorithm uses singular value decomposition to achieve lower dimension matrices which can approximate the conceptual similarity.

B. DOCUMENT VECTORS

Doc2Vec is originated from Word2vec [7], which is an artificial neural network that can transform (*embed*) words into vector space (*embedding*). The main idea is that the hidden layer of the network has fewer neurons than the input- and output layers, thus forcing the model to learn a compact representation. The novelty of Doc2Vec is that it can encode documents, not just words, into vectors containing real numbers.

C. TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY

TF-IDF is a basic technique in information retrieval. It relies on numerical statistics reflecting how important a word is for a document in a corpus. The frequency value is a metric that increases each time a word appears in the document but is offset by the frequency of the word in the whole corpus, highlighting specific words for each document.

D. RESULT REFINEMENT WITH ensemble$_N$ LEARNING

In our previous works [4]–[6] our experimental analysis led us to the conclusion that different ML techniques capture different similarity concepts. This means, that each examined technique can provide useful information, while generally, the desired code class appears close to the top of every similarity list. Thus, it should be possible to refine the obtained results a technique provides with another list that comes from a separate technique. The algorithm is very simple: only those code classes remain which are present in both similarity lists. Since every code class is ranked in the lists, we limit the search to the top N most similar ones, this way the algorithm will drop out the classes from the first list which are not amongst the top_N links of the second.

E. SOFT COMPUTED CALL INFORMATION

Since the listed techniques do not take class information into account, an additional simple filter can also be added. The following assumptions should be true in most cases: (1) the package of the class under test should either be the same as the test's or it should be imported in the test and (2) a valid target class should have a definition for at least one method name that is called inside the body of the test case. These criteria still do not guarantee a valid match.

Methods and imports are obtained from the Java files using regular expressions. These may differ for different programming languages by their different syntax.

F. EXTENDED NAMING CONVENTION EXTRACTION

The above presented techniques all result in a filtered list of soft computed links - i.e. there is no guarantee, that those are correct. Naming conventions, however, are known to produce traceability links with very high precision [3]. If a project lacks these good naming practices, naming conventions simply cannot be used in finding the correct matches. In this final approach, the naming convention is observed first. If it is applicable, it is accepted. Otherwise, the results of an IR-based approach (LSI, Doc2Vec, etc.) is considered.

Even though our experiments involved systems written in the Java programming language, the applied IR-based techniques mainly use the natural language part of the code, making the approach semi-independent from the chosen programming language. Nevertheless, language invariability cannot be guaranteed. These methods also depend on the habits of the developers. The naming conventions and the descriptiveness of the language of the natural text factors in a great deal in textual similarity. This is why it is also crucial that the developers possess sufficient education and experience to produce sufficiently clean source code. Furthermore, as it is visible with our current systems under test, systems with similar properties can produce vastly different results with the same methods.

Our experiments feature the extraction of program code from the systems under test using static analysis, obtaining different input representations, distinguishing tests from production code, textual preprocessing, and determining the conceptual connections between tests and production code. During our experiments, the Gensim [8] toolkit's implementation was used for all three textual methods. The initial static analysis that provides the text of each method and class of a system in a structured manner is achieved with the Source Meter [9] static source code analysis tool.

The proposed approach recommends classes for test cases 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 makes it a recommendation system, which provides the most similar parts of production code for each test case. Examining not only the most similar class but the top_N most similar ones has the benefit of highlighting the test and code relationship more thoroughly.

The proposed approach was evaluated on eight medium-sized open source projects written in Java, a further overview of these systems is available in Subsection II-I. In this paper, the models are not trained on plain source code, the feasible input representations are introduced in the next section.

```
boolean contains(Object target) {
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

FIGURE 2. An example method declaration, from which the AST was generated on Figure 3.

G. OPTIMAL INPUT REPRESENTATION

It is evident that the exact contents of the input are of crucial importance. In this section, we briefly describe the representations of code snippets (classes or methods) used in this work. A code representation is the input of a machine learning algorithm that computes the similarity between distinct items. Abstract Syntax Trees (AST) were utilized to form a sequence of tokens from the structured source code. An AST is a tree that represents the syntactic structure of the source code, without including all details like punctuation and delimiters. For instance, a sample Abstract Syntax Tree is displayed in Figure 3 which was constructed from the source code of Figure 2. To better understand the advantages and best possible methods of using the AST, the paper describes experiments on five different code representations, of which four relies on AST information. The five chosen representations are described below. The five representations under evaluation were constructed according to our previous work and are some of the most widely used representations in other research experiments [10], constructed along the work of Tufano *et al.* [11].

1) SRC

Let us consider the source code as a structured text file. In this simple case, similar methods are used in the context of natural language processing. These techniques include the tokenization of sentences into separate words and the application of stemming. With natural language, the separation of words can be quite simple. In the case of source code, however, we should consider other factors as well. For instance, compound words are usually written by the camel case rule, while class and method names can be separated by punctuation. The definition of these separators are one of the main design decisions in this representation. For the current work words were split by the camel case rule, by white spaces and by special characters that are specific to Java (“(“, “[”, ”.”). The Porter stemming algorithm was used for stemming. This approach notably does not use the AST of the files, making it a truly only text-based approach.

2) TYPE

To extract this representation for a code fragment, an Abstract Syntax Tree has to be constructed. This process ignores comments, blank lines, punctuation, and delimiters. Each node of the AST represents an entity occurring in the source code. For instance, the root node of a class (CompilationUnit) represents the whole source file, while the leaves are the identifiers in the code. In this particular case, the types of AST nodes were used for the representation. The sequence of symbols was obtained by pre-order traversal of the AST. The extracted sequences have a limited number of symbols, providing a high-level representation.

3) IDENT

Every node in the Abstract Syntax Tree has a type and a value. The top nodes of the AST correspond to a higher level of abstraction (like statements or blocks), their values typically consist of several lines of code. The values of the leaf nodes are the keywords used in the code fragment. In this representation, these identifiers are used by traversing the AST tree and printing out the values of the leaves. The values of literals (constants) in the source code also might occur here, these are replaced with placeholders representing their type (e.g. an integer literal is replaced with the *<INT>* placeholder, while a string literal with *<STRING>*). The extracted identifiers contain variable names. In the current experiments, they were split according to the camel case rule popularly used in Java.

4) LEAF

In the previous two representations, distinct parts of the AST were utilized to get the input. This approach takes both the types and node values into account. Just as before, a pre-order visit is performed from the root. If the node is an inner node then its type, otherwise (when it is a leaf) its value is printed. This representation captures both the abstract structure of the

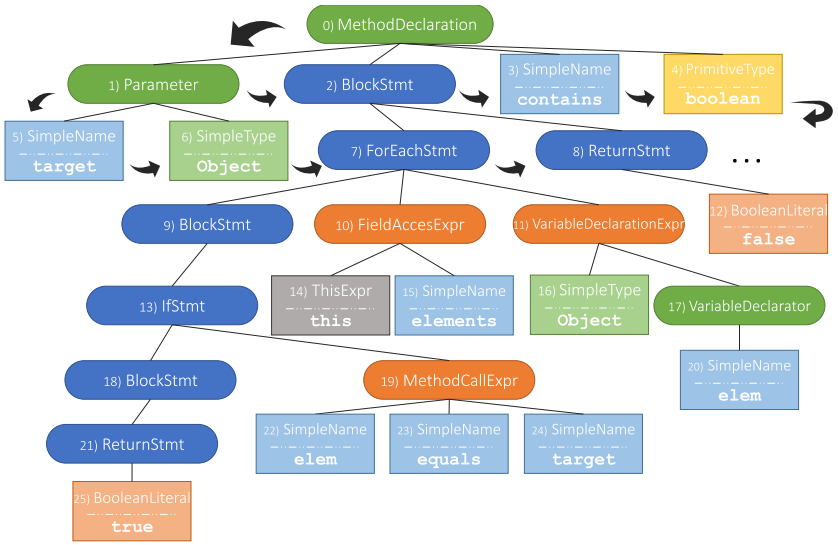

FIGURE 3. An Abstract Syntax Tree, generated from the example of Figure 2. The numbers inside each element indicate the place of the node in the visiting order. Leaves are denoted with standard rectangles (note that here the value and the type is also represented), while intermediate nodes are represented by rectangles with rounded corners.

AST and the code-specific identifiers. Considering the latter, these can be very unique and thus very specific to a class or a method.

5) SIMPLE

The extraction process is very similar to the previous one, except that in this case only values with a node type of *SimpleName* are printed out. These nodes occur very often, they constituted 46% of an AST on average in our experiments. These values correspond to the names of the variables used in the source code while other leaf node types like literal expressions or primitive types hold very specific information. Note that in the IDENT representation, the replacing of literals eliminated the AST node types of literal expressions. Only the modifiers, names, and types remained, thus becoming similar to this representation. With this representation, however, we do not exclude the inner structure of the AST.

H. EVALUATION PROCEDURE

In their 2009 evaluation, Van Rompaey and Demeyer [3] found that the naming conventions technique produced 100% precision in finding the tested class at each test case it was applicable to. The authors used a human test oracle consisting of 59 randomly chosen test cases altogether. These can be considered too few measuring points for proper generalization, but nevertheless, it is visible that naming conventions can identify the class under test in the overwhelming majority of the cases. Naming convention pairs can also be extracted automatically from method, class, and package names. Thus, one of our evaluation methods relies on the naming conventions technique.

Since naming convention habits may influence this, our approach was also evaluated on a human test oracle described in [12]. TestRoutes is a manually curated dataset that contains data on four of our eight subject systems, Commons Lang, Gson, JFreeChart and Joda-Time. It is a method-level dataset that classifies the traceability links of 220 test cases (70 from JFreeChart, 50 from each of the others). This information is also suitable for class-level evaluations, as this is a relaxed version of the same problem. The dataset lists the methods under test as focal methods (there can be multiple focal methods for a test case), as well as test and production context. Our current focus is on the classes of these focal methods. For JFreeCart and Joda-Time, the dataset specifically targeted test cases that were not covered with simple naming conventions, this will also be evident at our results. For the other systems, the dataset contains data on randomly chosen test cases.

The TestRoutes data was annotated by a graduate student familiar with software testing. The tests were not executed during the annotation process. The annotator worked in an integrated development environment, studied the systems' structure beforehand, and maintained regular communication with the researchers, addressing the arising concerns. The collected traceability links were inspected and validated by a researcher, with another researcher also verifying the links of at least ten test cases of each system.

A relatively simple yet sufficiently strict set of rules was applied in the naming convention based evaluation. Our NC-based evaluations were based on package hierarchy and exact name matching. This is further detailed in Subsection III-A, where this particular naming convention ruleset is referred to as PC (package + class).

The well-known precision metric was utilized to quantify our results. Precision is the proportion between correctly detected units under test (UUT) and all detected units

under test. It computes as

$$precision = \frac{relevantUUT \cap retrievedUUT}{retrievedUUT}$$

With such an evaluation, it is only possible to find one pair to each test case correctly. Our methods produce a list of recommendations in order of similarity. Every class is featured on this list. Thus, with our current evaluation methods, the customary precision and recall measures always coincide, which necessarily means that the F-measure metric would also have the same value. This is in accordance with the evaluation techniques commonly used for recommendation systems in software engineering. Because of this equality, we shall refer to our quantified results in the future as precision only.

I. SAMPLE PROJECTS

Our results were evaluated on multiple software systems and with multiple settings. These involved the following open-source systems: ArgoUML is a tool for creating and editing UML diagrams. It offers a graphic interface and relatively easy usage. Commons Lang is a module of the Apache Commons project. It aims to broaden the functionality provided by Java regarding the manipulation of Java classes. Commons Math is also a module of Apache Commons, aiming to provide mathematical and statistical functions missing from the Java language. Gson is a Java library that does conversions between Java objects and Json format efficiently and comfortably. JFreeChart enables Java programs to display various diagrams, supporting several diagram types and output formats. Joda-Time simplifies the use of date and time features of Java programs. The Mondrian Online Analytical Processing (OLAP) server improves the handling of large applications' SQL databases. PMD is a tool for program code analysis. It explores frequent coding mistakes and supports multiple programming languages.

TABLE 1. Size and versions of the programs used.

Program	Version	Classes	All Methods	Test methods
ArgoUML	0.35.1	2 404	17 948	554
C. Lang	3.4	596	6 523	2 473
C. Math	3.4.1	2 033	14 837	3 493
Gson	2.8.0	757	2 467	924
JFreeChart	1.0.19	953	11 594	2 239
Joda-Time	2.9.6	522	9 934	3 779
Mondrian	3.0.4.11371	1 626	12 186	1 546
PMD	5.6.0	1 608	9 242	825

The versions of the systems under evaluation, their total number of classes and methods, and the number of their test methods are shown in Table 1, while Figure 4 visually reflects these numbers. It has to be noted that several methods of the test packages of the projects have been filtered out as helpers since they did not contain any assertions.

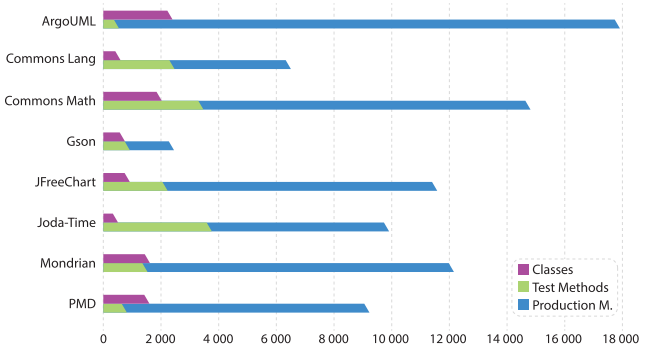

FIGURE 4. Properties of the sample projects used.

III. RESULTS

The current section evaluates the various approaches described in the previous section, featuring the results obtained from different representations and learning settings. First, various naming convention possibilities are overviewed with their applicability values determined via automatic extraction. Next, our experiments with the $ensemble_N$ approach are presented, where the best N value has been sought on NC-based and manual traceability links. Finally, the traceability approaches are compared to each other based on both NC and manual evaluation.

We note that production methods containing less than three tokens in their method bodies were filtered out since trivial and abstract production methods are not likely to be the real focus of a test.

A. APPLICABILITY OF NAMING CONVENTIONS

Naming conventions for tests are a vague term that can mean a multitude of various practices. The conventions are usually agreed on by the developers and written guidelines rarely even exist. They can also be only considered a mere good practice, and their use varies by teams or even individuals. As there can be various naming conventions, and their use is different in most systems, relatively vague criteria are needed to detect them in a versatile manner. Let us consider a few general criteria for our examination. These are presented in Figure 5. There are, of course, other possible criteria, including abbreviations or some other distinction for tests except the word "Test". Still, these seem to be the most intuitive and most popular naming considerations.

Let us consider some of the possible combinations of the listed criteria components. Figure 6 presents these. Some other viable combinations can also exist, which did not seem suitable for the unique distinction of test-code pairs. The criteria are ordered by strictness in a descending manner. While the stricter criteria produce more distinction between pairs, they are less versatile and are harder to uphold. Table 2 presents the extent to which the naming conventions were found to be applicable to the evaluated systems.

TABLE 2. The applicability of the naming conventions technique using different approaches.

Naming Criteria	ArgoUML	Commons Lang	Commons Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
PCM	14.91%	17.04%	12.50%	1.74%	32.60%	3.60%	6.57%	7.93%
PM	20.73%	19.80%	16.85%	2.18%	38.95%	10.09%	9.04%	8.43%
PCWM	19.82%	56.67%	32.19%	9.59%	49.53%	23.78%	11.51%	15.86%
PWCWM	21.27%	66.73%	37.52%	9.59%	50.92%	59.65%	12.09%	16.48%
PWM	33.45%	70.79%	45.42%	15.47%	58.64%	74.42%	31.01%	25.53%
M	28.91%	19.96%	21.16%	3.05%	40.15%	11.38%	12.22%	11.90%
PC	60.18%	84.58%	75.07%	26.14%	96.47%	36.80%	17.82%	58.36%
C	64.00%	84.58%	75.07%	27.89%	96.47%	37.30%	20.81%	66.91%
WM	74.00%	80.00%	81.53%	60.24%	61.28%	78.55%	73.34%	58.36%
PWC	75.09%	99.11%	88.06%	28.87%	97.05%	98.04%	21.52%	61.09%
WC	80.55%	99.11%	91.42%	44.77%	97.41%	98.04%	35.96%	72.12%

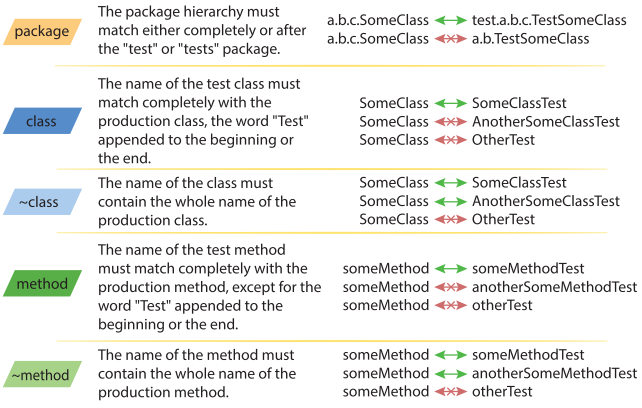

FIGURE 5. Various possible naming convention criteria components.

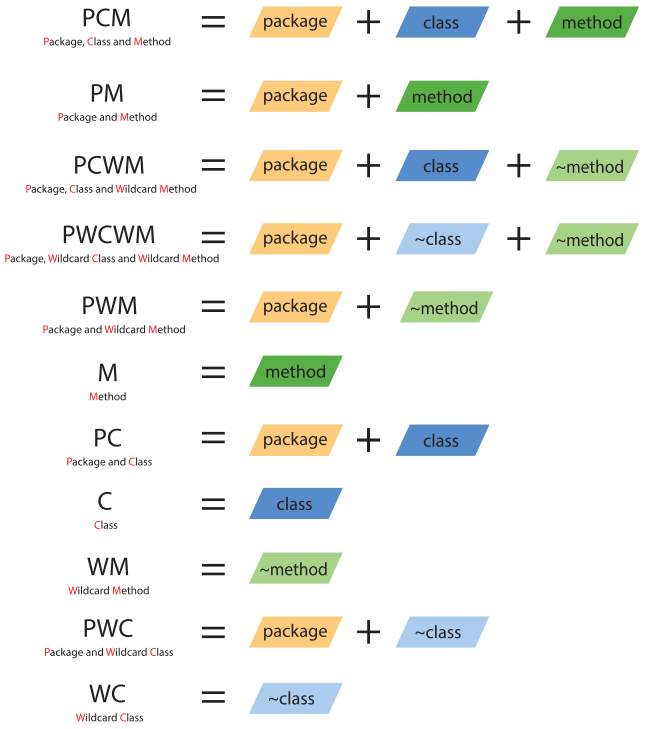

FIGURE 6. Some of the possible naming convention criteria in descending order of restrictiveness.

As it is visible from the results of the table, there is a significant jump in the applicability at the PC naming convention variant, which considers package hierarchy and an exact match to the name of the class. While the extent of the increase of applicability varies between systems, it is apparent that most of them produce only very few traceability links when method names are also considered. As our experiments at the current time feature class level test-to-code traceability, the further results of our paper will use PC as the default naming convention.

B. ENSEMBLE EXPERIMENTS

Figure 7 and Figure 8 show the results of our $ensemble_N$ learning approach. As one can see in the figures, the experiments were carried out using different N values: 50, 100, 200, and 400. These values only influence the size of each similarity list. If N is relatively big, then the filtering on the original similarity list (which originates from Doc2Vec) will not drop out many entries since many of the elements are present in the other two lists. In contrast, if N is a small number, the filtering is stricter since every similarity list contains only a limited number of entries. The previous argument can be further elaborated: if N is *big*, the resulting similarity list is going to rely mostly on the original one, while if it is *small*, the approach makes better use of the information from the other two approaches.

First, let us consider Figure 8, which visualizes the results from the sample projects measured via automatic naming convention extraction. The small flags on the top of the bars indicate the highest values for each system in their category (top_1, top_2, top_5). The flag's color is the same as its bar; a white flag means that the highest values are equal. Remarkably, no case was encountered where there were two or three highest values. In this experiment, the different source code representations are also considered. Looking at the figure, it is apparent that most of the flags appear at the IDENT representation. It is also worth mentioning that at the top_1 results, the $ensemble_{50}$ approach seems to produce the highest values. Considering multiple recommendations (top_2 and top_5), the situation is less obvious: $ensemble_{100}$ also seems to provide good results. $Ensemble_{400}$ seems to

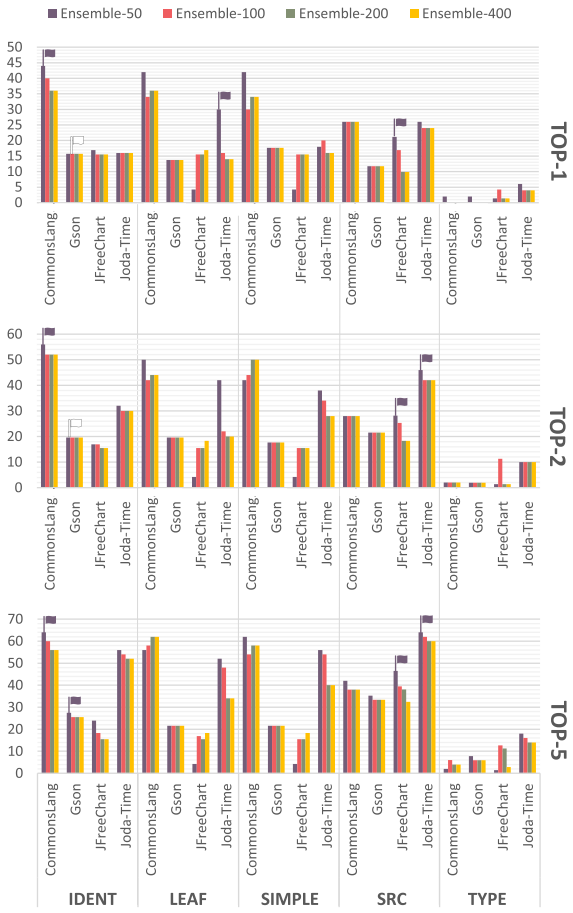

FIGURE 7. Results of the $ensemble_N$ learning approach measured on the manual dataset.

be less precise. It prevalent only in the case of Mondrian using the SIMPLE representation. The results on the manual dataset also reinforce this finding. In Figure 7, almost every flag belongs to the $ensemble_{50}$ approach, except in two cases, when it produced the same value as the others.

C. NC-BASED EVALUATION

Table 3 shows the top_1 results of different machine learning approaches, evaluated via naming conventions. Cells of color teal indicate the highest values for each system within a method, while cells of color violet indicate the overall top values. For the $Ensemble_N$, only those cases are listed where N = 50, since this setting seemed to be the most beneficial (for further discussion see Subsection III-B). The listed approaches correspond to the ones introduced in Section II. The notion *[approach]+**CG*** refers to filtering with our soft computed call information described in Section II-E.

D. EVALUATION ON MANUAL DATA

The results measured on the manual dataset are shown in Table 4. Similarly to the previous table, teal indicates the highest values within a method, while violet highlights the overall highest value. The left part of the table shows precision values of top-1 matches, while on the right side of the table, the top-5 results are listed. The top-5 results are always equal or higher than the top-1 numbers since there are more than one similar matches considered during the evaluation. Here, the results of different approaches are compared to the dataset's data, which contains manually curated traceability links on four of our subject systems. In this table, two additional rows are introduced. The first row shows the applicability of the naming convention (that was denoted PC in the previous subsection). These numbers depict the conventions' applicability to the specific test cases in the dataset, rather than the whole system. If naming conventions should be considered accurate, this value would intuitively correspond to the precision that could be achieved without any additional IR-based approach, only relying on the names. The last line's title contains the *NC* addition. Our method here first attempts to detect the link using naming conventions, and if it fails, the suggestion of Doc2Vec is considered. If the resulting precision values would be lower than before, that would either mean that the dataset is incorrect, or the naming conventions were misleading. It is also clear that if the results of this approach and the plain NC approach were equal, then the IR-based addition would be unnecessary. Eventually, none of these concerns were found to be reflected in Table 4. In fact, this approach produced the best results in almost every single case.

IV. DISCUSSION

A. NAMING CONVENTIONS HABITS

The previous section displayed some of the most common naming convention techniques and some data on how frequently they seem to have been utilized in the systems currently under our investigation.

Let us consider an example of how a perfect match would look like viewing all three of package hierarchy, class name, and method name. One such example for Commons Math is illustrated in Figure 9, which shows a test case (T) and the production method it is meant to test (P). If every single test case related to its method under test with such simplicity, test-to-code traceability would be a trivial task. Unfortunately, as it is visible from our applicability results, this is very far from reality.

According to our results, the names of test *methods* are much less likely to mirror the names of their production pairs correctly. Although our experiments only deal with eight open-source systems, it is highly probable that the developers of other systems also tend to behave similarly in focusing more on class-level naming conventions. WM (wildcard method) is obviously full of noise and accidental matches and cannot be considered seriously. PM (package+method) is a much more precise option but as it is visible it was found to be used in about every fifth case. One obvious reason for this can be that it is significantly harder to convey all the necessary information in method

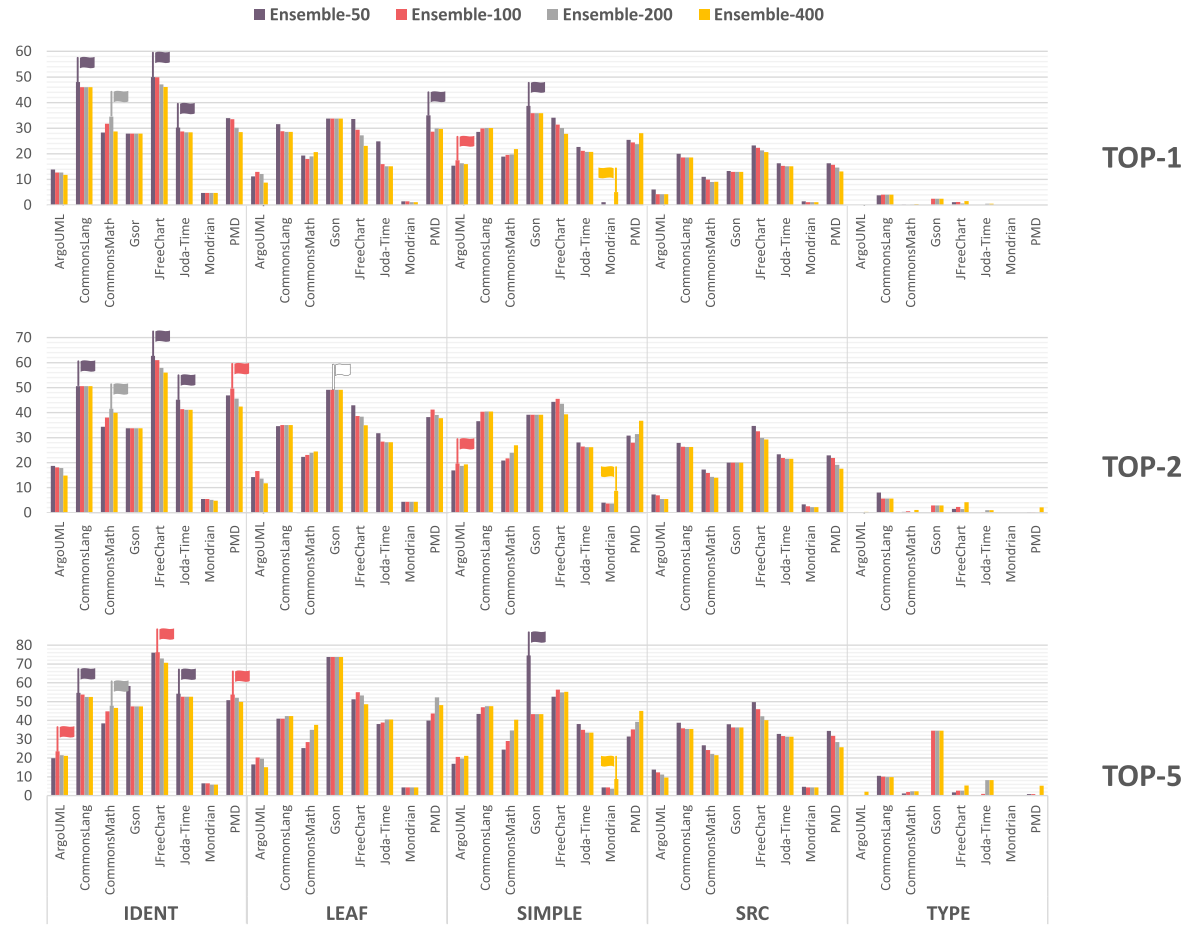

FIGURE 8. Results of the $ensemble_N$ learning approach using NC-based evaluation.

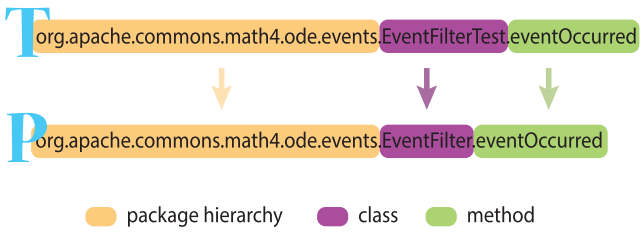

FIGURE 9. A trivial naming convention example from Commons Math.

names. Production method names should be descriptive and lead to an easy to understand and quick comprehension of what the method does. This is also true about the names of test cases, they should also refer to what functionality they are aiming to assess. Consequently, the names of the test cases would become rather long if they always aimed to contain both the name of the method or methods under test and also provide additional meaningful information about the test itself. It can also be tough to properly reference the method under test on method level by naming conventions only. Polymorphism enables the creation of several methods with identical names that perform similar functionalities with different parameters. These should be tested individually, and test names can have a hard time distinguishing these. The inclusion of parameter types can be a possible solution as performed in Commons Lang for example, at the test case *test_toBooleanObject_String_String_String_String*, testing the production method *toBooleanObject* that gets four String parameters. Our manual investigation shows that test methods are indeed more likely to be named after the functionality they mean to test rather than after single methods even if they only test one method. One method can also be tested by multiple test cases. Thus this is not a very surprising circumstance. It is apparent that naming conventions on the method level have to be more complicated, and their maintenance necessitates more work on the part of the developers. Thus, method-level naming solutions are likely to be a less valuable option in method-level test-to-code traceability. On the other hand, method-level traceability still requires proper class-level traceability. Thus, names should still be helpful.

Talking about *classes*, production class names seem to be mirrored more often in their test classes' names. This, however, still can be a highly unsteady habit depending on the system. While in Mondrian, production class names are

TABLE 3. Top-1 results featuring the different text-based models trained on various source code representations, evaluated using naming conventions. - highest value in a row - highest value in a column.

Method	Representation	ArgoUML	C. Lang	C. Math	Gson	JFreeChart	Joda-Time	Mondrian	PMD
Doc2Vec	IDENT	19.63%	82.16%	50.00%	45.83%	49.22%	41.43%	66.42%	37.15%
	LEAF	18.43%	61.00%	33.01%	47.92%	25.10%	20.79%	65.33%	42.04%
	SIMPLE	24.77%	67.91%	33.78%	47.50%	30.69%	26.26%	65.33%	34.82%
	SRC	7.85%	31.32%	15.46%	16.67%	22.64%	22.30%	21.53%	15.92%
	TYPE	0.60%	4.36%	0.78%	2.92%	2.36%	5.18%	0.00%	0.00%
LSI	IDENT	32.93%	66.08%	19.42%	30.83%	33.29%	35.04%	22.99%	19.96%
	LEAF	14.80%	23.11%	3.63%	7.08%	9.63%	16.12%	11.31%	7.80%
	SIMPLE	15.71%	21.48%	3.47%	4.37%	13.15%	6.69%	4.38%	11.46%
	SRC	19.64%	54.64%	24.36%	14.17%	21.48%	28.20%	31.02%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
TF-IDF	IDENT	35.95%	73.62%	35.78%	35.00%	45.65%	48.71%	73.72%	24.63%
	LEAF	32.63%	70.94%	37.33%	38.33%	48.93%	47.77%	66.79%	23.99%
	SIMPLE	28.70%	69.49%	33.08%	30.00%	44.30%	47.77%	72.26%	23.57%
	SRC	27.79%	51.51%	28.68%	18.75%	25.19%	31.08%	50.73%	22.29%
	TYPE	0.00%	0.48%	0.65%	4.58%	0.00%	0.50%	0.00%	0.00%
Ensemble-50	IDENT	13.89%	48.00%	28.27%	27.92%	50.00%	30.22%	4.75%	33.97%
	LEAF	11.18%	31.61%	19.29%	33.75%	33.56%	24.89%	1.45%	35.03%
	SIMPLE	15.41%	28.54%	18.93%	38.75%	34.01%	22.73%	1.01%	25.48%
	SRC	6.04%	20.00%	11.02%	13.33%	23.24%	16.33%	1.46%	16.35%
	TYPE	0.00%	3.79%	0.12%	0.00%	1.11%	0.00%	0.00%	0.00%
Doc2Vec+CG	IDENT	45.01%	83.14%	61.04%	85.83%	62.82%	43.02%	68.61%	54.14%
	LEAF	42.29%	72.66%	45.65%	44.17%	56.48%	34.10%	73.72%	52.23%
	SIMPLE	41.69%	71.89%	51.41%	52.92%	58.06%	24.32%	74.09%	51.80%
	SRC	32.33%	54.48%	29.62%	35.42%	37.36%	23.38%	47.08%	36.31%
	TYPE	3.63%	17.61%	13.22%	42.08%	10.46%	16.04%	41.24%	15.92%

only present in test names once in every fifth case, the same applies to 2160 of the 2239 test cases of JFreeChart. Thus, not surprisingly, it is evident that naming conventions depend on developer habits. The remaining test cases of JFreeChart were also examined, these are overwhelmingly cases where a specific type of charts or other higher-level functionalities are tested, and the test classes are named after these. These cases often depend on multiple production classes providing lower-level functionality.

Mirroring the *package-hierarchy* of the production code while composing tests is also a good practice. The little difference between the C (class) and PC (package+class) values in Table 2 shows that developers are very likely to uphold this. This convention is likely to be even more popular than naming matches. Package hierarchy matches are easier to maintain than names and are more convenient as they do not really require additional work from the developers. Even if multiple methods or classes are under tests, their packages only rarely differ. It could also be seen as another level of abstraction. Package hierarchy can only provide very vague clues about traceability links but can be suitable for the elimination of some of the false matches or at least presenting a warning sign about some matches. From the difference of matches found with C-PC and WC-PWC (wildcard class - packate + wildcard class), our manual investigation provided less conclusive results.

On the one hand, many systems contain some seemingly arbitrary exceptions to this rule that were most probably due to some design decision or modification in the production code that the structure of the tests has not followed yet. Gson, for example, has a ''gson'' package in its test structure that contains similarly named test classes to the ''internal'' package of the production code. Another example can be given from PMD, where there is an extra ''lang'' package in the hierarchy of the production code, that is not found at the test structure, even though all prior packages match. These are far from system-wide decisions as seen from the NC applicability percentages, but can be hard to detect by automatic means. On the other hand, some faulty matches do exist when not considering the package hierarchy. This can be seen at ArgoUML for instance, the ''Setting'' class of the production code can match with a lot of test classes if only names are taken into account, many of these would be faulty matches as the tests refer to different settings. Thus, matching packages is also far from a prerequisite in real-live systems. Like any other convention, developers make exceptions even if they visibly strive to uphold them at different parts of the code.

Without the insights from the developers of a system, our analysis had to judge their choice and usage of naming conventions based solely on the names themselves. This, of course, can be sufficiently accurate but presents the danger of not managing to grasp the whole system of conventions they used, which can vary. Still, our findings should provide a relatively accurate picture of how naming conventions are used in real-life testing solutions.

RQ1 answer: Although serious differences can be observed between systems, method-level naming conventions are either complicated or entirely abandoned in most cases,

TABLE 4. Top-1 and top-5 results featuring the different text-based models and the applicability of NC on each project. Models were trained on 5 different source code representations. - highest value in a row - highest value in a column.

Method	Representation	Top-1				Top-5			
		C. Lang	Gson	JFreeChart	Joda-Time	C. Lang	Gson	JFreeChart	Joda-Time
NC	-	76.00%	26.00%	0.00%	0.00%	76.00%	26.00%	0.00%	0.00%
Doc2Vec	IDENT	58.00%	15.69%	15.49%	32.00%	62.00%	25.49%	15.49%	48.00%
	LEAF	30.00%	13.73%	11.27%	20.00%	52.00%	21.57%	15.49%	52.00%
	SIMPLE	15.69%	17.65%	14.08%	16.00%	48.00%	21.57%	16.90%	52.00%
	SRC	16.00%	9.80%	12.68%	32.00%	42.00%	29.41%	30.99%	54.00%
	TYPE	4.00%	1.96%	11.27%	4.00%	22.00%	3.92%	11.27%	10.00%
LSI	IDENT	34.00%	17.65%	4.23%	10.00%	68.00%	5.64%	5.63%	44.00%
	LEAF	12.00%	7.84%	4.23%	2.00%	34.00%	23.53%	5.63%	28.00%
	SIMPLE	4.00%	5.88%	4.23%	2.00%	30.00%	23.53%	5.63%	24.00%
	SRC	34.00%	17.65%	12.68%	20.00%	70.00%	37.25%	23.94%	58.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
TF-IDF	IDENT	30.00%	19.61%	4.23%	46.00%	76.00%	31.37%	5.63%	70.00%
	LEAF	30.00%	19.61%	4.23%	44.00%	76.00%	33.33%	5.63%	70.00%
	SIMPLE	28.00%	21.57%	4.23%	44.00%	72.00%	33.33%	5.63%	72.00%
	SRC	38.00%	19.61%	23.94%	12.00%	78.00%	43.14%	25.35%	68.00%
	TYPE	4.00%	0.00%	0.00%	0.00%	8.00%	64.71%	0.00%	14.00%
Ensemble-50	IDENT	44.00%	13.73%	4.23%	6.00%	52.00%	23.53%	4.23%	10.00%
	LEAF	13.73%	13.73%	4.23%	10.00%	38.00%	19.61%	4.23%	14.00%
	SIMPLE	14.00%	15.69%	4.23%	2.00%	40.00%	19.61%	4.23%	8.00%
	SRC	7.84%	11.76%	11.27%	12.00%	36.00%	17.65%	28.17%	22.00%
	TYPE	2.00%	1.96%	0.00%	2.00%	8.00%	1.96%	0.00%	2.00%
Doc2Vec+CG	IDENT	58.00%	64.71%	16.90%	24.00%	76.00%	80.39%	23.94%	64.00%
	LEAF	54.00%	54.90%	18.31%	20.00%	72.00%	78.43%	33.80%	66.00%
	SIMPLE	50.00%	56.86%	25.35%	26.00%	76.00%	78.43%	45.07%	64.00%
	SRC	50.00%	56.86%	36.62%	32.00%	78.00%	82.35%	66.19%	74.00%
	TYPE	42.00%	47.05%	11.27%	6.00%	62.00%	74.51%	28.17%	24.00%
Doc2Vec+CG+NC	IDENT	76.00%	64.71%	16.90%	24.00%	86.00%	72.55%	23.94%	64.00%
	LEAF	78.00%	64.71%	18.31%	20.00%	84.00%	70.59%	33.80%	66.00%
	SIMPLE	78.00%	66.71%	25.35%	26.00%	84.00%	76.47%	45.07%	64.00%
	SRC	80.00%	66.67%	36.62%	32.00%	88.00%	78.43%	66.19%	74.00%
	TYPE	74.00%	64.71%	11.27%	6.00%	78.00%	76.47%	28.17%	24.00%

which means that their usefulness is negligible in a general extraction algorithm. Class-level naming conventions seem to be better regarded by developers, and there is a visible effort to uphold them. Our findings show them to be suitable for general use in automatic extraction algorithms. Matching package hierarchies do not provide precise results but seem to be at least as commonly used as class-level conventions. They are likely to be suitable for filtering out false-positive results in algorithms.

B. TRACEABILITY LINK RECOVERY TECHNIQUE IMPROVEMENTS

It is apparent at first sight that the teal cells are overwhelmingly located in the first rows in Table 3. Indeed, the IDENT source code representation seems to be prevalent: it reaches the highest values in 37 cases out of 48 (which is 77%). The violet cells appear only in the last vertical segment of the table, at the Doc2Vec+CG approach.

On the one hand, the $Ensemble_{50}$ approach produced better results than standalone techniques (Doc2Vec, LSI, TF-IDF) and the soft-computed call graph information even improved upon this. On the other hand, Doc2Vec supplemented with this call information resulted in the highest precision values. How is this even possible? The most probable explanation is that $Ensemble_N$ is a filter technique: the resulting similarity list is a reduced one compared to the original (especially when N is a relatively small number). Thus it can happen, that even before applying CG, the $Ensemble_{50}$ already dropped out some of the correct links.

According to the results of the table, IDENT seems the most precise approach. The only exception is the Mondrian project, where the SIMPLE representation appeared to perform best. The difference between IDENT and SIMPLE, however, is not remarkable. It is also worth mentioning that where IDENT is not predominant, SIMPLE was found best in 5 cases out of 11. Subsection II-G already stated that IDENT and SIMPLE are quite similar. This is also reflected in the results. In contrast, TYPE seems to produce weaker results with every single approach. LEAF is also less precise, probably because its inner structure shares a significant part with TYPE (LEAF is essentially the combination of IDENT and TYPE). From this, a conclusion can be made that the TYPE information of an AST holds less important information for the text-based test-to-code traceability task.

RQ2 answer: Our inspections concluded that Doc2Vec seems to be the best-performing standalone technique in the

field. Although combinations of different techniques can also boost the results, the textually extracted soft-computed call information is likely to boost IR-based approaches even more. In a scenario of combined techniques, call graphs can be a valid filter even for textual connections.

C. PERFORMANCE ON MANUAL DATA

Compared to the NC-based evaluation, the results captured on the manual dataset are less easy to interpret. As it is visible in Table 4, not every violet cell appears in the last row, only most of them. Let us first analyze the top-1 results which are shown on the left side of the table. At 3 out of 4 systems, the highest precision values are reached using Doc2Vec combined with the call information and naming conventions. The only exception is Joda-Time, where TF-IDF seems to be prevalent. In our previous work [4], a similar case has already been noted, where TF-IDF also provided the highest precision values. Even in these experiments, however, TF-IDF results are found to be much more variable than others, and this individual case is most probably a result of chance. It is visible, however, that doc2vec+CG still seems to have produced high precision values, and that applying naming conventions can further boost the approach. In the case of JFreeChart and Joda-Time, the results did not improve despite the added naming convention pairs. This is not surprising since as it is visible, the naming conventions were not applicable for any methods of these systems (as the dataset contained specifically such links by intention). It can, therefore, be stated that IR-based approaches can successfully supplement naming conventions while still maintaining their useful properties.

Looking at the right side of Table 4, the precision values are higher than before. It is quite self-evident since here the text-based models have a broader range to guess for the correct matches. While observing top-1 results, the best performing technique was not unanimous. Here, the highest precision values are all located in the last segments. While top-1 results varied in their precision, JFreeChart and Joda-Time having lower results than the other two systems, even a small number of additional candidate links has significantly contributed to the correctness of the matches. By further experiments, it was found that when considering top-10 or even top-20 results, a 100% match would not be uncommon either, though searching through a list of 10 artifacts is not a likely behavior of real-life developers. Thus, their everyday use of these would not be viable. Five results, however, could still make a simple recommendation system.

By studying the manual database, it can be observed that in the cases of projects where proper naming conventions were used, the traceability links can be extracted relying only on this information. However, for those projects which lack these good programming practices, IR-based techniques can find the correct links in a significant part of the cases. Comparing the results of the final Doc2Vec approach and NC itself, the precision values are higher by 28% on average. Even in the cases where some more complex, system-specific naming conventions were utilized, IR-based methods can provide great assistance. While there are likely to be some special cases in the practical use of every naming convention, the use of good programming practices can also improve the performance of text-based techniques which still rely on names in a more versatile manner.

From these results, the choice of an ideal input representation is a more elusive question than in the previous case. While at the NC-based comparison, the IDENT representation seemed prevalent against others, here the SRC representation seems to have produced the highest precision values. It is worth mentioning that among the representations that rely on AST information, SIMPLE performed best. At the Top5 values, it is also clear that the SRC representation won. The good results of SRC are also advantageous since SRC is a purely text-based representation. Since the Doc2Vec+CG+NC method features call information extracted via regular expressions, this is a viable option even without static analyzer tools. While both IDENT and SRC are shown to contribute valuable data, this difference between their relative performance on thousands of test cases of eight systems and the manual data on 220 test cases of four systems makes it harder to believe in a single best representation. Since the possible mistakes in the automatically gathered NC data and the less than ideal size of the manual dataset can both contribute to less than precise results in this respect, further research is still necessary for the question of best representation.

RQ3 answer: According to the manual data, Doc2Vec achieves the best results in most cases. In exceptional cases, however, other text-based techniques can still outperform it. The use of naming conventions and call information also tends to improve the results further. Naming conventions, if existing, are highly precise and can be supplemented with other IR-based techniques to achieve a more versatile text-based approach.

D. IMPLICATIONS

Our experiments show some simple implications for those who research and aim to build new test-to-code traceability solutions.

While naming conventions are reliable tools and are very precise if applied, they are harder to implement on the method level, and the source code generally contains fewer such connections that could be extracted via simple rules. However, packages and class names can imply the connections rather well, even for this level, even if method names are not as informative. Thus, naming conventions can be extremely useful on every level of traceability link extraction, and new extraction methods would most probably benefit from considering them.

Doc2Vec seems an important upgrade to the more mainstream semantic similarity techniques. While it is still somewhat more resource-intensive than LSI, the difference is not prominent, and just like the other techniques, Doc2Vec is also capable of providing real-time results of most similar parts of code for a test case. Thus, if a single textual technique should be considered, Doc2Vec seems to be the right choice.

The combination of Doc2Vec and other techniques can produce even better results. However, as it was seen that applied as filters, other textual techniques can drop out some of the valuable data, and even if they performed better together in separation, this combination could negatively impact the overall cooperation with other, non-semantic techniques. Call information, even if just gathered via regular expressions, tends to boost these techniques greatly. Combination with call graphs obtained via static or dynamic analysis could undoubtedly result in even better precision, as seen in current state-of-the-art solutions where the LCBA (last call before assert) technique is considered one of the most reliable methods.

Source code representations are less conclusive at the current time. IDENT seemed best in our previous and current NC-based evaluations, but manual data shows that SRC contributes most to the correct extraction. Thus, additional experiments are still required to announce a clear best representation. Nevertheless, these two are the most likely options.

V. 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 [13], [14]. Based on the study of Borg *et al.* [15], most of the traceability evaluations have been conducted on small bipartite datasets containing fewer than 500 artifacts, which is a severe threat to external validity. While data limitations still persist, the current paper's evaluation is conducted on eight software systems, using different oracles. While to the best of our knowledge, test-to-code traceability is not the most widespread topic amongst other recovery tasks, several well-known approaches aim to cope with this problem. Still, as yet, none of them has provided a perfect solution for the problem [3]–[6], [16], [17]. The current state-of-the-art techniques [18] rely on a combination of diverse methods - i.e. techniques based on dynamic slicing and contextual coupling. The use of textual information is common in these techniques. Our current work took a closer look at various textual similarity techniques, and combinations of these resulted in promising recovery precision.

In a recent work [19], authors presented TCtracer, a tool which combines an ensemble of new and existing techniques and exploits a synergistic flow of information between the method and class levels. The tool observes test executions and create candidate links between these artefacts and the tested artefacts. It then assigns scores (which are used to rank the candidates) to the candidate links. These scores are calculated using the combination of eight test-to-code traceability techniques including four string-based techniques, two statistical, call-based techniques, Last Call Before Assert (LCBA) and Naming Conventions (NC). Although this and our work share many common factors, there are significant differences. First of all, our technique does not rely on information based on test-execution. Secondly, the two rankings are fundamentally different: our work relies on IR techniques (and refine these using various approaches, with an initial static analysis), while White *et al.* calculate the ranking scores based on formulas defined in the paper. Finally, we also researched different ways of representing the source code.

The utilization of structural information has also occurred in other works [18], [20], [21]. In their 2015 work, Ghafari *et al.* [22] also employed structural information. Here, the main goal was to identify traceability links between test cases and methods under test, which is still not a mainstream topic in the field, as most methods aim for production classes. The proposed approach correctly detects focal methods under test automatically in 85% of the cases. Bouillon *et al.* leveraged a failed test case to find the location of errors in source code [23]. To link the tests to the production code, they built the static call graph of each test method and annotated each test with a list of methods which may be invoked from the test. The use of structural information also occurs in other extraction methods, feature extraction for instance, where it was shown that its combination with LSI is capable of producing good results [24]. In the current paper, structural information was used in several source code representations. Call information was also utilized, even though it was extracted only from the text. Even so, it was found a valuable addition as a filter.

Like LSI, TF-IDF is also a text-based model commonly used in the software engineering domain. This technique was, for instance, used by Yalda *et al.* [25] to trace textual requirement elements to related textual defect reports, and by Hayes *et al.* [26] in the after-the-fact tracing problem. In requirement traceability, the use of TF-IDF is so widespread, that it is considered a baseline method [27]. Text-based models are still very popular in the requirement traceability task also, they are incorporated in several recent publications [28], [29]. Our experiments covered LSI and TF-IDF as standalone techniques and also as a refinement for Doc2Vec, which was shown in our previous work [4] to produce higher quality results.

In our findings, the use of document embeddings resulted in the highest precision values. Word2Vec [30] gained a lot of attention in recent years and became a very popular approach in natural language processing. Calculating similarity between text elements using word embeddings became a mainstream process [10], [11], [31]–[34]. Doc2Vec [7] is an extension of the Word2Vec method dealing with whole documents rather than single words. Although not enjoying the immense popularity of Word2Vec, its use is still prominent in the scientific community [35]–[38].

The use of recommendation systems is widespread in the field of software engineering [1], [39], [40]. Presenting a prioritized list of most likely solutions seems to be a more resilient approach even in traceability research [5], [6].

Because of the numerous benefits of tests, developers tend to create a lot of them even though it is challenging to determine what new tests to add to improve the quality of a test suite. Since 100% coverage is often infeasible, several new approaches have been proposed for interpreting

coverage information. For instance, Huo *et al.* [41] introduced the concepts of direct coverage and indirect coverage, that address these limitations. In addition, several other challenges are present in general software testing [42], like coherent testing, test oracles and compositional testing. The more challenges are solved, and the more the community understands about testing in general, the better test-to-code traceability results can become [43]. The current paper also aimed to shed some light on class and method naming habits which can lead to a better understanding of testing in real-life software systems.

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.* [18], [21] incorporate textual analysis for more precise recovery. Jin *et al.* in [32] presented a solution that uses deep learning and word embeddings to incorporate requirements artifact semantics and domain knowledge into the tracing solution. The authors evaluated their approach against LSI and VSM (Vector Space Model). They found that their neural-based approach only outperforms these when the tracing network has a large enough training data which is hard to obtain. Other works also explore the use of word embeddings to recover traceability links [32], [44], [45]. Our current approach differs from these in many aspects. To begin with, we make use of different similarity concepts and further refine these with structural information. Next, our document embeddings are computed 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 it), and there was no additional natural language corpus.

VI. THREATS TO VALIDITY

Although our experiments were conducted with the intention of providing a large-scale evaluation and a relatively deep comprehension of current textual methods, some threats to the validity of the derived conclusions still have to be mentioned. While naming conventions are considered a very precise source of information, they have clear limitations. Thus, our automatically-collected evaluation data may contain some errors and is likely to miss at least some valid links. Although manual data is usually considered best, naming conventions enabled us to assess hundreds of tests for each system and even thousands for most. On the other hand, our manual dataset used for the evaluation is limited in size. Thus, noise in the data could cause discrepancies in the results. This could be tackled by the inclusion of additional manual data, which will hopefully be more widely available in the future.

Our experiments only covered systems written in the Java language. This is a significant limitation as Java differs greatly from several other popular programming languages. Even the structure of the code can show severe differences. Popular naming conventions can vary in these circumstances, new viable combinations could be constructed, and others could become less relevant. This also reflects a great amount on the source code representations. Even the text and even variable names could be susceptible to such a difference. On the other hand, textual methods, building on semantic information rather than program structure, are still the most likely to retain their properties this way.

The experiments were conducted on JUnit tests. The JUnit framework is one of the trail-blazers of current software testing and is extremely popular among developers. Still, it is easy to see that other tests could perform differently when subjected to the experiments. Even in this, however, semantic information should be the least affected as it does not rely on a specific structure or specific forms of assertion statements.

Similarly to the difference in programming languages, the size of the systems could also influence the results. Our systems under evaluation are all medium-sized open-source systems. There is no guarantee that small or large systems would perform the same way, even though the question of proper traceability is probably easier for small systems. The same questions can arise about the domains of the systems, which could also affect traceability. It is visible that systems vary significantly in their properties. An average value of precision is thus hard to pinpoint, it is easier to compare techniques to each other. Our experiments covered more than 1.25 million code lines to provide a large-scale investigation, significantly more than our previous inquiries.

Our experiments with naming conventions and even the source code representations represent the options we found most viable. There might be many more naming conventions that could be applied to some systems with great success, even with automated extraction. As there are usually no descriptions about naming conventions for software systems, finding these and judging their usefulness is highly complex. Our experiments considered some of the most simple and widely used conventions. There seems to be a balance between complete precision and easy usage in naming conventions. Our experiments also attempted to investigate this, building our subsequent experiments on a middle way that seemed widely applicable but still precise for our current level.

VII. CONCLUSION

The current paper showcased our experiments with the textual aspect of aiding test-to-code traceability. Two mainstream techniques, reliance on naming conventions and information retrieval were investigated, new ideas, experiments and observations were given on their possible improvements and combination opportunities. The paper presented an in-depth investigation of the naming convention habits of developers via experiments with eight open-source systems and nine possible combinations of generalizable and simple rules. This experiment revealed that package and class level conventions are generally followed with at least a moderate effort, but method level conventions, although present in every system, are less generally upheld. Besides our evaluation on manual data, an automatic extraction was also used for further evaluation, relying on package and class level conventions. The six investigated traceability link extraction methods were

evaluated with five different source code representations. From these, the identifier-centric (IDENT) representation that utilizes abstract syntax trees came out on top in the overwhelming majority of the cases during the naming convention based evaluation but the text-centric (SRC) representation proved more precise when compared to a limited amount of manual data. Call information retrieved via regular expressions was found to contribute significantly to the results when used as a filtering technique for Doc2Vec. Although the use of LSI and TF-IDF also seems a good candidate for the same purpose, the combination of Doc2Vec and the call information was found to produce the best results. While properly defined and upheld naming conventions can yield extremely precise traceability links, their use is still limited. Automatic recovery of naming conventions, however, can very easily benefit from the addition of other text-based techniques, together constituting a versatile semantic technique that can still be used in combination with other mainstream methods.

REFERENCES

[1] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Test. Anal.* New York, NY, USA: ACM, Jul. 2016, pp. 165–176.

[2] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, vol. 11, May 2018, pp. 789–799.

[3] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 209–218.

[4] V. Csuvik, A. Kicsi, and L. Vidács, *Evaluation of Textual Similarity Techniques in Code Level Traceability* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11622. Cham, Switzerland: Springer, 2019, pp. 529–543.

[5] A. Kicsi, L. Tóth, and L. Vidács, "Exploring the benefits of utilizing conceptual information in test-to-code traceability," in *Proc. 6th Int. Workshop Realizing Artif. Intell. Synergies Softw. Eng.*, 2018, pp. 8–14.

[6] V. Csuvik, A. Kicsi, and L. Vidacs, "Source code level word embeddings in aiding semantic test-to-code traceability," in *Proc. IEEE/ACM 10th Int. Symp. Softw. Syst. Traceability (SST)*, May 2019, pp. 29–36.

[7] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2. Red Hook, NY, USA: Curran Associates, 2013, pp. 3111–3119.

[8] *Gensim Gensim*. Accessed: 2019. [Online]. Available: https://radimrehurek.com/gensim/

[9] (2019). *SourceMeter*. [Online]. Available: https://www.sourcemeter.com/

[10] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, Aug. 2016, pp. 87–98.

[11] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, vol. 18, May 2018, pp. 542–553.

[12] A. Kicsi, L. Vidács, and T. Gyimóthy, "TestRoutes: A manually curated method level dataset for test-to-code traceability," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Jun. 2020, pp. 593–597.

[13] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[14] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of traceability links between software documentation and source code," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 15, pp. 811–836, Oct. 2005.

[15] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: A systematic mapping of information retrieval approaches to software traceability," *Empirical Softw. Eng.*, vol. 19, no. 6, pp. 1565–1616, Dec. 2014.

[16] N. Kaushik, L. Tahvildari, and M. Moore, "Reconstructing traceability between bugs and test cases: An experimental study," in *Proc. 18th Work. Conf. Reverse Eng.*, Oct. 2011, pp. 411–414.

[17] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "SCOTCH: Test-to-code traceability using slicing and conceptual coupling," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2011, pp. 63–72.

[18] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *J. Syst. Softw.*, vol. 88, pp. 147–168, Feb. 2014.

[19] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.* New York, NY, USA: ACM, Jun. 2020, pp. 861–872.

[20] A. Panichella, C. Mcmillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "When and how using structural information to improve IR-based traceability recovery," in *Proc. 17th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2013, pp. 199–208.

[21] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "Evaluating test-to-code traceability recovery methods through controlled experiments," *J. Softw., Evol. Process*, vol. 25, no. 11, pp. 1167–1191, Nov. 2013.

[22] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *Proc. IEEE 15th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2015, pp. 61–70.

[23] P. Bouillon, J. Klinke, N. Meyer, and F. Steimann, *EZUNIT: A Framework for Associating Failed Unit Tests With Potential Programming Errors* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 4536. Berlin, Germany: Springer, 2007, pp. 101–104.

[24] H. Eyal-Salman, A.-D. Seriai, C. Dony, and R. Al-msie'deen, "Recovering traceability links between feature models and source code of product variants," in *Proc. VARiability Workshop Variability Modeling Made Useful Everyone (VARY)*. New York, NY, USA: ACM, 2012, pp. 21–25.

[25] S. Yadla, J. H. Hayes, and A. Dekhtyar, "Tracing requirements to defect reports: An application of information retrieval techniques," *Innov. Syst. Softw. Eng.*, vol. 1, no. 2, pp. 116–124, Sep. 2005.

[26] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Improving After-the-Fact tracing and mapping: Supporting software quality predictions," *IEEE Softw.*, vol. 22, no. 6, pp. 30–37, Nov. 2005.

[27] S. K. Sundaram, J. H. Hayes, and A. Dekhtyar, "Baselines in requirements tracing," in *Proc. Workshop Predictor Models Softw. Eng. (PROMISE)*, vol. 30, no. 4. New York, NY, USA: ACM, 2005, p. 1.

[28] J. M. Florez, "Automated fine-grained requirements-to-code traceability link recovery," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng, Companion (ICSE-Companion)*, May 2019, pp. 222–225.

[29] T. Hey, "INDIRECT: Intent-driven requirements-to-code traceability," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion)*, May 2019, pp. 190–191.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2, Dec. 2013, pp. 3111–3119.

[31] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 38–41.

[32] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 3–14.

[33] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2016, pp. 127–137.

[34] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 438–449.

[35] Z. Zhu and J. Hu, "Context aware document embedding," *CoRR*, vol. abs/1707.01521, 2017. [Online]. Available: http://arxiv.org/abs/1707.01521

[36] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," *CoRR*, vol. abs/1507.07998, 2015. [Online]. Available: http://arxiv.org/abs/1507.07998

[37] S. Wang, J. Tang, C. Aggarwal, and H. Liu, "Linked document embedding for classification," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.* New York, NY, USA: ACM, Oct. 2016, pp. 115–124.

[38] 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, Mar. 2015.

[39] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Softw.*, vol. 27, no. 4, pp. 80–86, Jul. 2010.
[40] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer-Verlag, 2014.
[41] C. Huo and J. Clause, "Interpreting coverage information using direct and indirect coverage," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2016, pp. 234–243.
[42] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Proc. Future Softw. Eng. (FOSE)*, May 2007, pp. 85–103.
[43] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Trans. Rel.*, vol. 63, no. 4, pp. 913–926, Dec. 2014.
[44] T. Zhao, Q. Cao, and Q. Sun, "An improved approach to traceability recovery based on word embeddings," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 81–89.
[45] 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 *Proc. 38th Int. Conf. Softw. Eng.* New York, NY, USA: ACM, May 2016, pp. 404–415.

ANDRÁS KICSI received the master's degree in computer science from the University of Szeged, in 2017. He is currently an Assistant Research Fellow with the University of Szeged. His research interests include the uses of natural language in several innovative research and industrial topics, especially in artificial intelligence applications. His current research interests include machine understanding of medical reports and images, the author identification of general text, the facilitation of test-to-code traceability, and software product line adoption. He is taking part in various projects of the MTA-SZTE Research Group on Artificial Intelligence and has a passion for education.

VIKTOR CSUVIK received the M.Sc. degree in computer science from the University of Szeged, where he is currently pursuing the Ph.D. degree. His research interests include automatic program repair and machine learning in general. He is also interested in the topics of data visualization and software product lines. Besides his academic activities at the MTA-SZTE Research Group on Artificial Intelligence, he worked several years as a Software Developer with the Department of Software Engineering, University of Szeged. Moreover, he has teaching duties at the University of Szeged, and is also gaining industry software development experience as an Intern at an International Software Company.

LÁSZLÓ VIDÁCS received the Ph.D. degree from the University of Szeged, Hungary, in 2010. He is currently a Senior Research Fellow and the Deputy Head of the MTA-SZTE Research Group on Artificial Intelligence, while his research is strongly connected to the Department of Software Engineering, University of Szeged. He received best paper awards from the IEEE ICPC 2009 and the IEEE CSMR-WCRE 2014 conferences. He continuously takes part in R&D&I projects based on academia/industry collaborations with leading national and international companies. Building on program analysis and testing research background, he is recently working at the intersection of artificial intelligence and software engineering with special interest in natural language processing and deep learning.