TestRoutes: A Manually Curated Method Level Dataset for Test-to-Code Traceability

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

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

Tibor Gyimóthy
gyimothy@inf.u-szeged.hu
Department of Software Engineering, University of Szeged
MTA-SZTE Research Group on Artificial Intelligence, University of Szeged

ABSTRACT

High test-to-code traceability can be an important aspect of quality assurance and can contribute to bug localization and code maintenance. Several existing techniques and a considerable effort from the scientific community already made significant advances in the field. Despite this, readily accessible data on traceability links is very scarce. To contribute to related research, we present a manually curated test-to-code traceability dataset containing the traceability information on 220 test cases. This method-level data was gathered from 4 open-source software systems written in the Java language, distinguishing not only focal information on test cases but also highlighting the utilized helper methods on both the test and production aspects of code. The data includes more than 2000 of such method classifications.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

traceability, testing, test-to-code, dataset

ACM Reference Format:
András Kicsi, László Vidács, and Tibor Gyimóthy. 2020. TestRoutes: A Manually Curated Method Level Dataset for Test-to-Code Traceability. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3379597.3387488

1 INTRODUCTION

Ensuring software quality is a vital topic with a vast amount of scientific and industrial background. One of its main aspects is software testing which aims to uncover the faults that lie within the software. Extensive testing is considered good practice throughout the world. In the case of a larger software system, staggering amounts of tests can be created as a byproduct of the development. It is not rare for a system to incorporate tens of thousands of tests.

Traceability in software engineering is the ability to trace work items across various software artifacts [Antoniol et al. 2002; Marcus et al. 2005]. Having a large number of tests, even telling which part of code a test aims to evaluate can mean a considerable difficulty. This problem, tracing tests to their units under test, is called test-to-code traceability. In many cases, it means more than just considering the methods a test case calls, which can be abundant. It needs to consider the intent of the author of the test, to point at specific parts of the code that the author meant to test. While good coding practices like following some kind of naming conventions can help this process, even these can not solve every problem. Considering an already existing system, retrospectively enforcing such conventions can be bordering on the impossible. Thus, automatic extraction methods are necessary to uncover the traceability links.

Several authors dealt with this problem on class level [Kicsi et al. 2018a; Rompaey and Demeyer 2009] but the main advantages of proper traceability information lie in method level identification, which could ease the development and open new doors for bug localization. At method level, we can talk about focal methods [Ghafari et al. 2015] which are the methods the tests were written to evaluate. The main challenge lies in finding these methods.

Lacking sufficiently certain focal information, the evaluation of the proposed methods is always difficult. Our dataset aims to ease this burden by providing manually annotated data that can contribute to the efforts of the community. This paper introduces the TestRoutes[1] dataset and describes its properties and structure. The dataset includes method level traceability links for 220 test cases. In addition to linked pairs of methods, we provide the context of each test both at test and production sides, involving more than 2000 manually discovered methods. Compared to data used in related research, this dataset is a big step forward in granularity, in volume and in supplying the context to the tests.

2 BACKGROUND

2.1 The Route of a Test Case

The dataset is named TestRoutes which alludes to the many different methods a test usually visits during its run. In Listing 1, we display Joda-Time's *org.joda.time.TestDateTime_Basics.testIsAfterNow()* test that asserts that the *org.joda.time.base.AbstractInstant.isAfterNow()* method runs properly. Figure 1 illustrates this process.

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

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387488

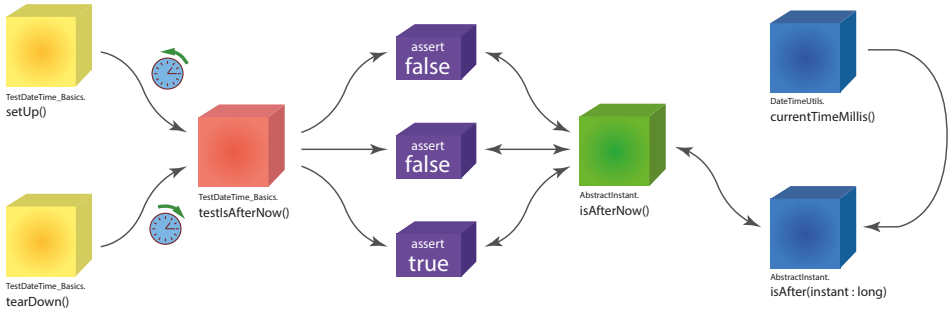

Figure 1: The testing process of a test case of Joda-Time

```
public void testIsAfterNow() {
  assertEquals(false,
    new DateTime(TEST_TIME_NOW - 1).isAfterNow());
  assertEquals(false,
    new DateTime(TEST_TIME_NOW).isAfterNow());
  assertEquals(true,
    new DateTime(TEST_TIME_NOW + 1).isAfterNow());
}
```

Listing 1: An example on a JUnit test case of Joda-Time

The method aims to provide a boolean value signifying whether the instance it belongs to resembles a date and time in milliseconds after our current date and time. To evaluate this, the test calls the method three times on instances set to be less, equal and greater than our current date and time. This naturally increases throughout development, thus the real present moment is not appropriate for testing this method. The test uses a helper method, *setUp()* to set several different global settings of the current run of Joda-Time like the current date, time, locale and time zone to predefined values (2002-06-09). The *tearDown()* helper method sets these settings back to the original values after the testing is finished. When the isAfterNow() method receives a call, it immediately makes a call to the org.joda.time.base.AbstractInstant.isAfter() method which expects a *long* value as a parameter, the milliseconds it should compare to. This should be the current date and time, so it calls the *org.joda.time.DateTimeUtils.currentTimeMillis()* method. The real check is inside the isAfter() method, the decision is made and is given back, eventually reaching the assertion. If everything works properly, the process decides that only a moment in the future is recognized as occurring in the future.

2.2 Systems Explored

The dataset contains information on four open-source software systems written in the Java programming language. The version and size information of these systems are displayed in Table 1. Please refer to our readme on GitHub for further details. Commons Lang is a module of the Apache Commons project that aims to add new class manipulation possibilities to Java. Joda-Time provides a replacement for the standard Java date and time classes which was widely used before Java SE 8. JFreeChart helps displaying various diagrams in several different formats in Java programs. Gson supports the serialization and deserialization of Java objects. JFreeChart is the largest of the systems but Joda-Time has the most tests, more than a third of its total methods are tests.

Table 1: Versions and size of the systems explored

System	Version	Classes	Prod. Methods	Test methods
Commons Lang	3.4	596	4 050	2 473
Joda-Time	2.9.6	522	6 155	3 779
JFreeChart	1.0.19	953	9 355	2 239
Gson	2.8.0	757	1 543	924

3 DATASET

3.1 Structure

Our dataset distinguishes four different method level roles in the testing process. The **test cases** are the methods aiming to assess a specific part of the software. They often contain one or more assertion statements and make method calls to conduct the evaluation. The test cases often enlist the aid of **helper** methods for setup or modularity purposes. These methods are customarily well separated from the production part of the source code. The **focal** methods (the units under test) are the methods the test case aims to test. There can be multiple focal methods for a test case, or even none, for instance, if the test case evaluates a property of a whole class. Focal methods can rarely achieve their full functionality by themselves, test cases often make several other calls. These **contextual** calls and the calls made by the focal method can also be important. Note that the call graph was not considered for the tests, only those contextual production methods are marked whose correct behavior was found important in the current context, vitally contributing to the pass of a test case. To our knowledge, there are no precise definitions for the distinction of focal and contextual methods. These are reliant on the probable intent of the author that is hard to measure objectively. Tests and helper methods are easier to distinguish as helpers usually do not make assertions.

```
test_case_id;method_id;method_role
JT-000058;JT-000059;focal
JT-000058;JT-000060;contextual
JT-000058;JT-000061;contextual
JT-000058;JT-000062;helper
JT-000058;JT-000063;helper
```

Listing 2: An extract from *Joda-Time_routes.csv*

The dataset includes two types of files. The traceability links can be found in the **route files** (_routes.csv extension). An extract displaying the very same traceability links we have seen before in

Figure 1 for the testIsAfterNow() test case can be seen in Listing 2. The values are separated by semicolons. The first column marks the test case, the second column contains a connected method while the third column describes the nature of the connection, this is either *focal*, *contextual* or *helper*. The methods are referenced by unique identifiers that can be resolved using the data files.

The **data files** (_data.csv extension) contain more information on the referenced methods. An example with the same methods of Joda-Time is displayed in Listing 3. Note that the separator here is not the same as in the other file's case, since semicolons can occur in qualified names. The first column is the unique identifier of the method while the following columns contain the header, qualified name, starting and ending position of each method in this order. Positions describe the line numbers of the methods in their files.

3.2 Data

The presented dataset contains information on four systems. The number of methods in the dataset can be seen in Table 2 for each system, grouped by category. Some interesting observations can be made from this, for example, Joda-Time seems to use a large number of test helpers while Commons Lang uses very few and also appears to often test more than one method with a test case. The number of production context methods also differ seriously.

Table 2: The number of methods by each role assigned

System	Test Methods	Focal Methods	Prod. Context Methods	Test Helpers
Commons Lang	50	89	91	11
Joda-Time	50	54	312	101
JFreeChart	70	79	430	58
Gson	50	55	102	30

In our previous work [Csuvik et al. 2019a,b; Kicsi et al. 2018a] we utilized naming conventions to extract class level traceability links for evaluation purposes. This can be achieved by automatic means but only in cases where known conventions were applied. While the test cases of Commons Lang and Gson were chosen randomly from the whole systems, in the case of Joda-Time and JFreeChart the random choices in tests were limited to the test cases not applicable to some very simple naming convention rules. Moreover, since only these 70 tests were found not to be covered by naming conventions in JFreeChart, it is possible to cover the whole system on class level by combining our data and naming conventions. Most test cases are not very isolated in their calls, they often make use of the same methods. In Figure 2, a graph representation of the connections is shown displaying the data on the Joda-Time system. The green nodes represent the test cases, the white nodes mark every other method within the dataset, and the edges correspond to the links in the data. The majority of the graph forms a loosely-connected subgraph with only a few isolated test cases.

The manual annotation has been conducted by an undergraduate student. The main objective was capturing the author intent of the tests relying on code comprehension, method calls, comments, naming conventions and descriptive names. Review sessions were held during the work with one of the authors and the annotator,

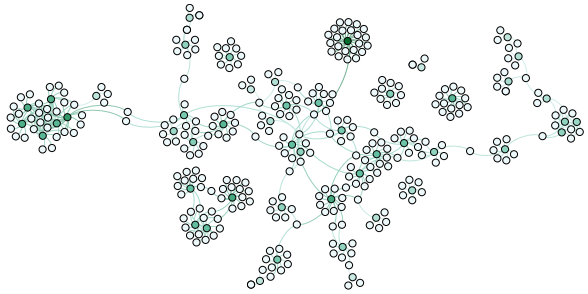

Figure 2: A graph of connections between test cases (green colored nodes) and other methods from Joda-Time in the dataset

addressing the guidelines and the encountered questionable cases. The author reviewed the results after the annotation and corrected the mistakes in both format and content. The test cases that were found to be highly subjective or required a very deep understanding of the system were exchanged to different test cases. About 15 tests were replaced this way. In ongoing related research, this data is already under use in evaluating various traceability techniques. For this, another researcher also validated the data, checking 10 test cases of each system he found no obvious mistakes. The gathering and inspection of the data were conducted in a Java development environment. The codebase was studied beforehand by all participants, the structure and purpose of the code were known, but the tests have not been analyzed or run. After the manual annotation, the systems were submitted to static analysis with the SourceMeter[SourceMeter 2018] tool to obtain the normalized versions of the qualified names and line information.

While most researchers seem to prefer evaluation with manually annotated data [Ghafari et al. 2015; Rompaey and Demeyer 2009], there are serious difficulties in its collection. The resulting data is only rarely available publicly [Kicsi et al. 2018a; Qusef et al. 2014]. While most recent research efforts still evaluate on class level, TestRoutes provides method-level granularity and in addition to focal information also highlights the test and production context, which to our knowledge haven't been included yet in other dataset.

Let us address the threats to the validity of the data. Firstly, the production context and focal information can be rather subjective, thus different annotators might differ in judgment even though comments and descriptive naming can imply the units under test with high probability. The random test case selection aims to cover a broad perspective but statistically significant statements can not really be made about the whole systems based on them. We also make no statements about the real naming convention coverage of the systems, many of the test cases of JFreeChart and Joda-Time in the data can also be covered by some form of conventions, we only considered a simple method which we found sufficiently precise.

3.3 Possible Uses

The knowledge of real traceability links facilitates test-driven development and improves software evolution. It can enable seamless integration between continuous code changes and unit tests, and serve as an important source of system documentation [Ghafari

```
"method_id","method_header","method_qualified_name","method_start_line","method_end_line"
"JT-000058","void testIsAfterNow()","org.joda.time.TestDateTime_Basics.testIsAfterNow()V","452","456"
"JT-000059","boolean isAfterNow()","org.joda.time.base.AbstractInstant.isAfterNow()Z","332","334"
"JT-000060","boolean isAfter(long instant)","org.joda.time.base.AbstractInstant.isAfter(J)Z","322","324"
"JT-000061","long currentTimeMillis()","org.joda.time.DateTimeUtils.currentTimeMillis()J","71","73"
"JT-000062","void setUp()","org.joda.time.TestDateTime_Basics.setUp()V","109","117"
"JT-000063","void tearDown()","org.joda.time.TestDateTime_Basics.tearDown()V","119","127"
```

Listing 3: An extract from *Joda-Time_data.csv*

et al. 2015]. Our dataset serves these goals, it can provide a common ground in evaluating test-to-code traceability solutions, potentially contributing to a better understanding of different traceability link extraction methods and lead to their best combination. Data is highly necessary for all future improvements.

The dataset is mainly intended for test-to-code traceability research evaluation purposes. It provides more detailed data than the currently available alternatives and can also be used in method-level evaluations. As method-level traceability links are very rarely recoverable relying on naming conventions, automatic options are highly limited in this case. Manual data is likely to be the most accurate evaluation option. The dataset is also suitable for class-level assessment. While the test cases of Commons Lang and Gson represent random tests from the systems, the test cases of Joda-Time and JFreeChart contain random tests that are not covered by naming conventions, also enabling evaluations on how the presence of naming conventions influences a method. As some traceability links can be extracted automatically with naming conventions, we are planning to also include these in the future. We plan to provide a MySQL database of the data for easier querying.

4 RELATED WORK

Traceability in software engineering research usually means the discovery of traceability links from requirements or related natural text documentations towards the source code [Antoniol et al. 2002; Marcus et al. 2005]. Recovery methods have been investigated between several types of textual documents like bug descriptions [Rath et al. 2018] and even tests [Kaushik et al. 2011]. In these methods conceptual analysis [Diaz et al. 2013; Panichella et al. 2013] is widely applied for fault localization [Moreno et al. 2014], test-prioritization [Saha et al. 2015], feature analysis [Kicsi et al. 2018b] and traceability link recovery between, for example, tests and requirements [Marcus and Maletic 2003].

Test-to-code traceability is an intensively studied topic [Mader and Egyed 2012; Parizi 2016; Parizi et al. 2014] as well. Individual ways have been proposed like plugins integrated into development environments [Philipp Bouillon, Jens Krinke, Nils Meyer 2007] and also methods relying on static or dynamic analysis [Sneed 2004] or analyzing the co-evolution of the code [Vidács and Pinzger 2018]. Call graphs, the information in method or class names and timestamps have also been successfully utilized in this process. Rompaey et al. [Rompaey and Demeyer 2009] used three systems to evaluate the effectiveness of 6 different recovery techniques. Qusef et al. [Qusef et al. 2010] improved the last call before assert technique with data flow analysis, relying highly on data dependencies. In their follow-up works [Qusef et al. 2011, 2014] dynamic slicing is used to increase the number of identified connections and precision is maintained using the latent semantic indexing (LSI) technique. In our previous paper [Kicsi et al. 2018a], we provided a deeper analysis of the LSI method as a traceability link recovery technique at class level. Ghafari et al. [Ghafari et al. 2015] proposed an algorithm using program slicing and call information at method level.

The evaluation of the proposed approaches in test-to-code link recovery is usually accomplished using manually collected links. Rompaey et al. [Rompaey and Demeyer 2009] used 71 randomly selected tests in their evaluation in class level recovery. Similarly, the study of Qusef et al. [Qusef et al. 2014] is evaluated at class level, the authors provided 358 traceability links in their online appendix. Kicsi et al. [Kicsi et al. 2018a] used naming conventions as a standard, and provided data for 5 open source projects including thousands of class level traceability links. The algorithm proposed by Ghafari et al. [Ghafari et al. 2015] is evaluated at method level using 50 manually produced links. Currently, we are aware of only two datasets publicly available: a class level manually curated dataset from Qusef et al. [Qusef et al. 2014] and a class level dataset that relies on automatically mined data of naming conventions from Kicsi et al. [Kicsi et al. 2018a]. Our proposal contains 2000 method classifications for 220 tests at the method level and also provides the potentially important context both in the test suite and in the production code. We argue that it is important to consider test-to-code traceability links together with their closely related neighborhood.

5 CONCLUSIONS

Test-to-code traceability provides the target of test cases in the production code. State-of-the-art algorithms are typically evaluated using a relatively small number of manually collected links. In addition, only very few of these links are available, which hampers the comparative evaluation of novel methods. In this paper, we described a manually curated dataset, which contains traceability links for 220 test cases at method level granularity from 4 open-source Java programs. The dataset includes not only the traceability links but also highlights the context of the test and production methods. The whole dataset consists of more than 2000 categorizations of methods that were manually examined. The data is also available at https://doi.org/10.5281/zenodo.3741674.

ACKNOWLEDGMENTS

This work was supported in part by the ÚNKP-19-4-SZTE New National Excellence Program and grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002), by the Hungarian Government. László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences. We acknowledge the help of Alex Oláh.

REFERENCES

G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28, 10 (oct 2002), 970–983.

Viktor Csuvik, András Kicsi, and László Vidács. 2019a. Evaluation of Textual Similarity Techniques in Code Level Traceability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 11622 LNCS. 529–543.

Viktor Csuvik, Andras Kicsi, and Laszlo Vidacs. 2019b. Source code level word embeddings in aiding semantic test-to-code traceability. In *Proceedings - 2019 IEEE/ACM 10th International Workshop on Software and Systems Traceability, SST 2019*. 29–36.

Diana Diaz, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Silvia Takahashi, and Andrea De Lucia. 2013. Using code ownership to improve IR-based Traceability Link Recovery. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 123–132.

Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 61–70.

Nilam Kaushik, Ladan Tahvildari, and Mark Moore. 2011. Reconstructing Traceability between Bugs and Test Cases: An Experimental Study. In *2011 18th Working Conference on Reverse Engineering*. IEEE, 411–414.

András Kicsi, László Tóth, and László Vidács. 2018a. 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* (2018), 8–14.

András Kicsi, László Vidács, Viktor Csuvik, Ferenc Horváth, Arpád Beszédes, and Ferenc Kocsis. 2018b. Supporting Product Line Adoption by Combining Syntactic and Textual Feature Extraction. In *International Conference on Software Reuse, ICSR 2018* (Madrid, Spain). Springer International Publishing.

Patrick Mader and Alexander Egyed. 2012. Assessing the effect of requirements traceability for software maintenance. *IEEE International Conference on Software Maintenance, ICSM* (2012), 171–180.

Andrian Marcus and Jonathan I. Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. *25th International Conference on Software Engineering, 2003* (2003), 125–135.

Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. 2005. Recovery of Traceability Links between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering* (2005), 811–836.

Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*. IEEE, 151–160.

A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 199–208.

Reza Meimandi Parizi. 2016. On the gamification of human-centric traceability tasks in software testing and coding. In *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 193–200.

Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. 2014. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability* 63 (2014), 913–926.

Friedrich Steimann Philipp Bouillon, Jens Krinke, Nils Meyer. 2007. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In *Agile Processes in Software Engineering and Extreme Programming*. Vol. 4536. Springer Berlin Heidelberg, 101–104.

Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2011. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 63–72.

Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2014. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software* 88 (2014), 147–168.

Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. 2010. Recovering traceability links between unit tests and classes under test: An improved method. In *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 1–10.

Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing requirements and traceability information to improve bug localization. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, New York, New York, USA, 442–453.

Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *European Conference on Software Maintenance and Reengineering, CSMR*. IEEE, 209–218.

Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 268–279.

H.M. Sneed. 2004. Reverse engineering of test cases for selective regression testing. In *European Conference on Software Maintenance and Reengineering, CSMR 2004*. IEEE, 69–74.

SourceMeter 2018. SourceMeter Webpage. https://www.sourcemeter.com/.

László Vidács and Martin Pinzger. 2018. Co-evolution Analysis of Production and Test Code by Learning Association Rules of Changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2018)* (Campobasso, Italy). IEEE, 31–36.