Methods2Test: A dataset of focal methods mapped to test cases

Michele Tufano, Shao Kun Deng, Neel Sundaresan, Alexey Svyatkovskiy
Microsoft
Redmond, WA, USA
Email:{mitufano, shade, neels, alsvyatk}@microsoft.com

ABSTRACT
Unit testing is an essential part of the software development process, which helps to identify issues with source code in early stages of development and prevent regressions. Machine learning has emerged as a viable approach to help software developers generate automated unit tests. However, generating reliable unit test cases that are semantically correct and capable of catching software bugs or unintended behavior via machine learning requires large, metadata-rich, datasets. In this paper we present Methods2Test: a large, supervised dataset of test cases mapped to corresponding methods under test (i.e., focal methods). This dataset contains 780,944 pairs of JUnit tests and focal methods, extracted from a total of 91,385 Java open source projects hosted on GitHub with licenses permitting re-distribution. The main challenge behind the creation of the Methods2Test was to establish a reliable mapping between a test case and the relevant focal method. To this aim, we designed a set of heuristics, based on developers’ best practices in software testing, which identify the likely focal method for a given test case. To facilitate further analysis, we store a rich set of metadata for each method-test pair in JSON-formatted files. Additionally, we extract textual corpus from the dataset at different context levels, which we provide both in raw and tokenized forms, in order to enable researchers to train and evaluate machine learning models for Automated Test Generation. Methods2Test is publicly available at: https://github.com/microsoft/methods2test

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.
• Computing methodologies → Neural networks.

KEYWORDS
datasets, software testing

ACM Reference Format:
Michele Tufano, Shao Kun Deng, Neel Sundaresan, Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In 19th International Conference on Mining Software Repositories (MSR ’22). May 23–24, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528009

1 INTRODUCTION
Writing test cases is a critical part of software development yet often neglected by developers. While software testing is widely acknowledged as an effective step towards identifying bugs early in the development process, developers tend to prioritize the introduction of new features [8] over writing test cases. Automated software testing aims to fill this gap, by providing tools that facilitate the accurate generation, selection, and execution of tests.

Specifically, in the context of unit test cases, such as EvoSuite [10], Randoop [17], and Agitar [3]. These well-established techniques rely on search-based software engineering, code analysis, program synthesis, and constraint solving to suggest candidate test cases. These approaches often aim at maximizing traditionally accepted metrics such as code coverage (line and branch coverage) and mutation score. The automatically generated test suites can be used for regression testing purposes, providing a relative degree of confidence that the changes performed by developers did not introduce regressions in the system.

While these techniques undoubtedly provide a valuable contribution for developers, several empirical studies have uncovered weaknesses and limitations of the generated test cases, such as unsatisfactory code quality [11, 18, 20], poor fault-detection capability [21], and the inability to adequately meet the software testing needs of industrial developers [4, 22]. Moreover, from the developers’ perspective, these test cases often appear as machine-generated code, and thus, hard to read, understand, and maintain [7, 12]. These limitations may be due to the fact that existing techniques focus mainly on testing metrics such as code coverage and mutation score when generating test cases, rather than learning from real-world, developer-written tests.

Recent years have witnessed the proliferation of machine learning (ML) techniques aiming at solving different tasks within the software engineering (SE) realm. Examples of these tasks include automating developers’ activities, such as: bug-fixing [5, 27], fault localization [28], writing comments [13] and documentation [6], code reviews [25], and merge conflict resolution [24]. This represents a fundamental shift away from maximizing code metrics and towards learning from real-world examples, extracting patterns from large amount of publicly available data.

These ML-based techniques require large datasets in order to effectively learn from examples. However, these datasets need to balance the necessity for a large number of examples, with the quality constraints and requirements. As for many other cases, the concept of “garbage in, garbage out” applies to these ML-based approaches, which need to be trained on high quality data, since noise could hinder the learning process. Thus, several curated datasets have been proposed for different SE tasks, allowing researchers to train and evaluate their ML-based techniques. Examples of such
datasets include Defects4J [14], Bugs2Fix [27], and CodRep [5] for automated bug-fixing, NASA Metrics Data Program (MDP) [23], PROMISE [2], and the unified dataset [9] for defect prediction, Landfill [19] for code smell detection, and many others.

In the field of automated test generation, however, there is a noticeable absence of a large, curated dataset of real world test cases. Additionally, in order to effectively train an ML-based approach to generate test cases, the dataset cannot be a simple collection of tests, rather it needs to expose traceability links between tests and their corresponding methods under tests (i.e., focal methods).

To fill this gap, we propose Methods2Test: a large dataset of test cases mapped to their corresponding focal methods. This dataset contains 780k pairs of JUnit tests and focal methods, extracted from a total of 91k Java open source projects hosted on GitHub. In order to establish a reliable mapping between a test case and the relevant method under test, we designed a set of heuristics, based on developers’ best practices in software testing, such as class and method naming conventions.

To facilitate and enable further analysis even beyond the scope we initially intended, we store a rich set of metadata for each method-test pair in JSON-formatted files, including information about project, file, class, and methods related to the test case. Additionally, we extract parallel textual corpus from the dataset at different context levels, which we provide both in raw and tokenized forms, in order to allow researchers to train and evaluate ML models for Automated Test Generation. Methods2Test data and scripts are publicly available at: https://github.com/microsoft/methods2test

2 DATASET GENERATION

This section provides the details on how we created Methods2Test. We begin by describing the steps for mining test cases, and mapping them to corresponding focal methods (Sec. 2.1). Then we extract the different levels of focal contexts surrounding the focal method, which provides a useful representation for a test case generation model (Sec. 2.2). Finally, we illustrate how we organize and store the dataset (Sec. 2.3).

2.1 Test Case Extraction & Mapping

The goal of this stage is to mine test cases and corresponding focal methods (i.e., the method tested by the test case) from a set of Java projects. We select a 91K sample of all the public GitHub Java repositories declaring an open source license, which have been updated within the last five years, and are not forks.

First, we parse each project to obtain classes and methods with their associated metadata. Next, we identify each test class and its corresponding focal class. Finally, for each test case within a test class, we map it to the related focal method obtaining a set of mapped test cases.

Parsing. We parse each project under analysis with the treewriter parser [1]. During the parsing, we automatically collect metadata associated with the classes and methods identified within the project. Specifically, we extract information such as method and class names, signatures, bodies, annotations, and variables. The parsed code is used to identify test cases and corresponding focal methods, as well as augmenting the focal methods with focal context.

Finding Test Classes. In this stage, we identify all the test classes, which are classes that contain a test case. To do so, we mark a class as a test class if it contains at least one method with the @Test annotation. This annotation informs JUnit that the method to which it is attached can be run as a test case.

Finding Focal Classes. For each test class we aim to identify the focal class which represents the class under test. To this aim, we employ the following two heuristics, in a sequence:

- **Path Matching**: best practices for JUnit testing suggest placing code and corresponding test cases in a mirrored folder structure. Specifically, given the class src/main/java/Foo.java the corresponding JUnit test cases should be placed in the class src/test/java/FooTest.java. Our first heuristic tries to identify the folder where the focal class is defined, by following the path of the test class but starting with the src/main folder (i.e., production code).
- **Name Matching**: the name of a test class is usually composed of the name of the focal class, along with a "Test" prefix or suffix. For example, the test case for the class Foo.java would probably be named FooTest.java. Thus, following the path matching heuristic, we perform name matching to identify the focal class by matching the name of the test case without the (optional) "Test" prefix/suffix.

Finding Focal Method. For each test case (i.e., method within a test class with the @Test annotation) we attempt to identify the corresponding focal method within the focal class. To this aim, we employ the following heuristics:

- **Name Matching**: following the best practices for naming classes, test case names are often similar to the corresponding focal methods. Thus, the first heuristic attempts to match the test cases with a focal method having a name that matches, after removing possible Test prefix/suffix.
- **Unique Method Call**: if the previous heuristic did not identify any focal method, we compute the intersection between (i) the list of method invocations within the test case and (ii) the list of methods defined within the focal class. If the intersection yields a unique method, then we select the method as the focal method. The rationale behind this approach is as follows: since we have already matched the test class with the focal class (with very high confidence heuristics), if the test case invokes a single method within that focal class, it is very likely testing that single method.

Mapped Test Cases. The result of the data collection phase is a set of mapped test cases, where each test case is mapped to the corresponding focal method. It is important to note that we discard test cases for which we were not able to identify the focal method using our heuristics. We designed these heuristics to be based on testing best practices, and obtain a correct mapping with very high confidence. This will likely exclude test cases that have been automatically generated, which may not follow testing best practices.

We collect an initial set of 887,646 mapped test case pairs. From this set, we exclude duplicates, remaining with a total of 780,944 unique mapped test case pairs. Next, we split the dataset into training (~80% - 624,022 pairs), validation (~10% - 78,534 pairs), and test
2.2 Focal Context

The goal of this phase is to construct an input which contains the necessary information that an ML-based model could leverage to automatically generate correct and useful test cases. Intuitively, the focal method (i.e., the method under test) represents the core information to feed to a model. However, additional contextual information can provide important clues for the model to better understand the focal method nature and its context, improving the likelihood of generating test cases that compile and properly test the focal method.

We build different versions of the code input representation – with diverse degree of focal context – with the aim of providing to researchers and practitioners, a variety of corpora, of different lengths, that can be used to train their models. We begin with the core information (i.e., focal method) and iteratively add contextual information such as class name, constructors, other method signatures, and fields.

Figure 1 provides an overview of the different levels of context we generate for the focal method add in the Calculator class. The left side corresponds to the textual representation, while the right side delineates the context which is indicated with a focal context ID, which we describe in the following:

- **fm**: this representation incorporates exclusively the source code of the focal method. Intuitively, this contains the most important information for generating accurate test cases for the given method.

- **fm+fc**: this representations adds the focal class name, which can provide meaningful semantic information to the model.

- **fm+fc+c**: this representation adds the signatures of the constructor methods of the focal class. The idea behind this augmentation is that the test case may require instantiating an object of the focal class in order to properly test the focal method.

- **fm+fc+c+m**: this representations adds the signatures of the other public methods in the focal class. The rationale which motivated this inclusion is that the test case may need to invoke other auxiliary methods within the class (e.g., getters, setters) to set up or tear down the testing environment.

- **fm+fc+c+m+f**: this representation adds the public fields of the focal class. The motivation is that test cases may need to inspect the status of the public fields to properly test a focal method.

While constructing these representations we face two opposing goals: (i) include as many tokens as possible, given their powerful expressiveness discussed above (ii) keep a concise representation that fits into GPU memory, allowing anyone to train their model.

Intuitively, having a representation that includes many tokens from the focal context allows a model to attend to different parts of the input and leverage these information to generate a correct and meaningful test case. On the other hand, irrelevant tokens could represent noise for the learning process, which could lead to worse performances, as well as wasting GPU memory that could be used for more informative tokens.

It is important to highlight that in our representation, the order of inclusion of a particular focal context, for example the constructors’ signatures (fm+fc+c) before other methods’ signatures (fm+fc+c+m), is important, since the textual representation could be truncated if it exceeds 1024 tokens (i.e., maximum sequence length in most Transformer models).

This order of inclusion has been defined by the authors based on their understanding and intuition of the meaningful clues for test case generation within the focal class. We empirically evaluated these design decision in our previous work [26].

2.3 Organization & Storage

Methods2Test is publicly available on GitHub [16]. The repository makes use of the Git large file storage (LFS) service. Git LFS works by replacing large files in the repository with tiny pointer files.

We organize Methods2Test in three main folders: dataset, corpus, scripts. The dataset folder contains the test cases mapped to their corresponding focal methods, along with a rich set of metadata. The dataset is stored as JSON files for each individual method-test pair. Each JSON file contains metadata at repository-, class-, and method-level for both the test case and the focal method. The repository data include:

- **id**: unique identifier of the repository in the dataset

| Set          | Repositories | Mapped Test Cases |
|--------------|--------------|-------------------|
| Training     | 72,188       | 624,022           |
| Validation   | 9,104        | 78,534            |
| Test         | 10,093       | 78,388            |
| **Total**    | **91,385**   | **780,944**       |

(~10% - 78,388 pairs) sets. We performed this split by carefully taking into account possible data leakage. Specifically, during the split we enforce the constraint that any two data points belonging to the same repository cannot be placed in two different sets (e.g., one in training and the other in test). That is, all the data points belonging to the same repository will be placed in the same set.

Table 1 reports the details of the dataset split, with number of repositories and mapped test cases.

Figure 1: Focal Context

```
// Focal Class
public class Calculator {
    // Focal Method
    public float add(float op1, float op2) {
        float result = op1 + op2;
        this.previousValue = this.screenValue;
        this.screenValue = result;
        return result;
    }

    // Constructors
    Calculator();
    Calculator(float value);

    // Public Method Signatures
    public float subtract(float op1, float op2);
    public float multiply(float op1, float op2);
    public float divide(float op1, float op2);
    public void reset();
    public void revertLastOperation();
    public float getScreenValue();
    public float getPrevScreenValue();

    // Public Fields
    public float screenValue;
    public float prevScreenValue;
}
```
 JSON format supports the augmentation of the focal context with intended. Several empirical studies could be performed on this data, leveraging the traceability between test cases and functional code. Additionally, the availability of rich metadata in training of models of different sizes (e.g., smaller models on smaller datasets, to extract the test cases and identify the corresponding focal method. Defects4J represents an invaluable infrastructure for software testing. The most related dataset we found is TestRoutes [15], where the authors manually curated a test-to-code traceability dataset containing the traceability information on 220 test cases. While we share a similar goal, Methods2Test was intended to be a large-scale repository of test cases to enable training of ML-based techniques for automated test generation. Thus, we designed heuristics to perform such mapping, rather than relying on manual analysis.

Defects4J [14] offers a collection of reproducible bugs, as well as the triggering tests for each of these bugs. The link between the bug and the triggering test could potentially be used to recover the focal method. Defects4J represents an invaluable infrastructure for APR techniques, however, in terms of test cases, they only belong to 17 Java projects, while we mine tests from thousands of open source projects hosted on GitHub.

6 CONCLUSIONS

We presented Methods2Test: a large dataset of 780k JUnit test cases mapped to their corresponding focal methods (i.e., method under test), extracted from 91k Java open source projects hosted on GitHub. We described the heuristics, based on testing best practices, to extract the test cases and identify the corresponding focal methods. We collect a rich set of metadata for each method-test pair, and store them in JSON-formatted files. Additionally, we extract textual corpus from the dataset at different context levels, which we provide both in raw and tokenized forms, in order to enable researchers to train and evaluate ML models for Automated Test Generation. Methods2Test is publicly available at: https://github.com/microsoft/methods2test
REFERENCES

[1] [n. d.]. Tree-sitter. http://tree-sitter.github.io/tree-sitter.

[2] 2019. PROMISE. http://promisedata.org/.

[3] Agitar. 2020. Utilizing Fast Testing to Transform Java Development into an Agile, Quick Release, Low Risk Process. http://www.agitar.com/.

[4] M. Morein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Bene-felds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In 2017 IEEE/ACM 9th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 265–272.

[5] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monsieurs. 2019. Sequence: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering (2019).

[6] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT: multi-mode translation of natural language and Python code with transformers. arXiv preprint arXiv:2010.01250 (2020).

[7] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 107–118.

[8] Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In 2014 IEEE 24th International Symposium on Software Reliability Engineering. IEEE, 201–211.

[9] Rudolf Ferenc, Zoltan Toth, Gergely Ladanyi, Istvan Siket, and Tibor Gyzmathy. 2018. A public unified bug dataset for java. In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. 12–21.

[10] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 18th European conference on Foundations of software engineering. 416–419.

[11] Giovann Grano, Fabio Palomba, Davide Di Nucci, Andrea De Lucia, and Harold C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in object-oriented programming systems and applications companion. 815–816.

[12] Giovann Grano, Simone Scalabrino, Harold C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, 348–349.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, 290–29010.

[14] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis. 437–440.

[15] András Kicsi, Liszló Vádasz, and Tibor Gyzmathy. 2020. Testtroues: A manually curated method level dataset for test-to-code traceability. In Proceedings of the 17th International Conference on Mining Software Repositories. 593–597.

[16] Microsoft. 2020. methods2test. https://github.com/microsoft/methods2test.

[17] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. 815–816.

[18] Fabio Palomba, Davide Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: An empirical study. In 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST). IEEE, 5–14.

[19] Fabio Palomba, Davide Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An open dataset of code smells with public evaluation. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, 482–485.

[20] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In Proceedings of the 25th International Symposium on Software Testing and Analysis. 130–141.

[21] Gustavo HL Pinto and Silvia R Vergilio. 2010. A multi-objective genetic algorithm to test data generation. In 2010 22nd IEEE International Conference on Tools with Artificial Intelligence. Vol. 1. IEEE, 129–134.

[22] Sina Shamsbiri. 2015. Automated unit test generation for evolving software. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 1038–1041.

[23] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Maier. 2013. Data quality: Some comments on the nasa software defect datasets. IEEE Transactions on Software Engineering 39, 9 (2013), 1208–1215.

[24] Alexey Svyatkovskiy,Todd Mytkowicz, Negan Ghorbani, Sarah Fahkoury, Eliz-abeth Dunella, Christian Bird, Neel Sundaresan, and Shuvendu K. Lahiri. 2021. MergeBERT: Program Merge Conflict Resolution via Neural Transformers. CoRR abs/2109.00084 (2021). arXiv:2109.00084 https://arxiv.org/abs/2109.00084

[25] Rosalia Tufan, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 163–174.

[26] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE].

[27] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM) 28, 4 (2019), 1–29.

[28] W Eric Wong, Ruihui Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. IEEE Transactions on Software Engineering 42, 8 (2016), 707–740.