Property-based Testing within ML Projects: an Empirical Study

Cindy Wauters
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
cindy.suzy.wauters@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—In property-based testing (PBT), developers specify properties that they expect the system under test to hold. The PBT tool generates random inputs for the system and tests for each of these inputs whether the given property holds. An advantage of this approach over testing a set of manually defined example inputs is that it enables a higher code coverage.**

Machine learning (ML) projects, however, often have to process large amounts of diverse data, both for training a model and afterwards, when the trained model is deployed. Generating a sufficient amount of diverse data for the property-based tests is therefore challenging.

In this paper, we present the results of a preliminary study in which we examined a dataset of 58 open-source ML projects that have dependencies on the popular PBT library Hypothesis, to identify issues faced by developers writing property-based tests.

For a subset of 28 open-source ML projects, we study the property-based tests in detail and report on the part of the ML project that is being tested as well as on the adopted data generation strategies. This way, we aim to identify issues in porting current PBT techniques to ML projects so that they can be addressed in the future.

***Index Terms*—Testing Machine Learning Projects, Automated Testing, Property-based Testing**

I. Introduction

Property-based testing (PBT) can be a powerful tool to test software projects for unexpected behaviour. In contrast to example-based testing, a developer writes down a property that they expect to hold for the system under test. The PBT tool then generates inputs at random and tests for each of these inputs whether the given property is satisfied. For example, imagine a function `reverse`, that reverses any given string. A desired property of this function would be that calling `reverse` twice (e.g. `reverse(reverse(string))`), results in the original string. The PBT tool will randomly generate strings and test for each of these strings whether this is indeed the case, rather than using a single developer-provided example string.

In recent years, researchers have investigated how developers implement and use PBT in general-purpose software projects [3], [5], [6]. However, when it comes to testing machine learning (ML) projects, additional challenges can arise. These projects often require a lot of diverse data, rendering data generation more difficult or time-consuming, which encumbers their testing [4]. This is especially true for ML projects that deal with image classification or voice recognition. This was also discovered in studies by Sharma et al. who propose a *property-driven testing* technique (*MLCheck*) for ML classifiers [14] and ML models [15]. In an attempt to compare *MLCheck* to *Hypothesis* [9], a popular PBT library in Python, they found that after forty minutes of running the tests, Hypothesis was not able to generate inputs for the test cases due to the high number of features that were required to be generated. Furthermore, in large-scale ML projects, software developers and data scientists work alongside each other. Oftentimes, these groups of people have different skill sets and use different (testing) libraries and tools, thus complicating development and debugging [10]. Finally, ML projects often lack a clear test oracle. While the *oracle problem* [16] is a known issue in software testing, it is complex in ML projects due to, for example, a repetition of the training phase being able to yield a different behaviour [13]. Metamorphic testing (a subset of PBT) can address the problem. Here, *metamorphic relations* are used as property. A classic example of such a relation is the `sine` function, where it is difficult to know what the output of `sine(x)` will be for any randomly generated `x`, but testing whether the metamorphic relation `sine(`π`-x) = sine(x)` indeed holds for each generated `x` is easier.

Existing research has investigated how to transpose PBT and metamorphic testing to machine learning models and code (e.g., [11], [12], [14], [18]). In this paper, we empirically investigate how real-life, open-source projects are using PBT. We specifically look at the popular PBT library for Python, Hypothesis, which has over 7.4k stars on GitHub. It is also used by several popular Python projects such as PyTorch[1] and NumPy[2]. At the same time, a JetBrains survey from 2022, surveying 23,000 Python developers found that only 5% of developers use Hypothesis in their software [8]. This study thus aims to discover how developers are using property-based testing in open-source ML projects, how they handle the data generation aspect of PBT, and what difficulties they experience. By identifying these problems, they can become the subject of future work, rendering PBT more accessible and easier to use.

[1]https://pytorch.org/
[2]https://numpy.org/

```python
from hypothesis import given, strategies

@given(strategies.text())
def test_encode_decode(s):
  assert decode(encode(s)) == s
```

Listing 1: An example of a property-based test. Example from https://hypothesis.readthedocs.io/en/latest/quickstart.html

The remainder of this paper is structured as follows. First, PBT is introduced in Section II. Section III defines the research questions and describes data collection and analysis. Section IV presents results, and Section V identifies potential threats to the validity. Section VI discusses related work. Finally, conclusion and future work are discussed in Section VII.

II. Background

Property-based testing, first popularized by the QuickCheck [2] tool for Haskell, is an automated testing technique where the developer defines *properties* expected to hold for the system under test (SUT). The SUT can in this case be a single function (unit testing), multiple functions (integration testing), or the whole system (system testing). Alongside these properties, the developer specifies the expected input type that needs to be randomized (e.g., random strings, integers, lists, etc.). When the test is run, the PBT tool randomizes inputs based on the expected type, and checks for each of these inputs whether the tested property indeed holds. If for each of the generated inputs the property holds, the test passes. Otherwise, the test fails.

A simple example of a PBT (making use of the Hypothesis library) can be found in Listing 1, where an `encode` and a `decode` function are tested. A desired property of these two functions could be that the result of decoding an encoded string is identical to the original string. On line 1, we import `given` and `strategies` from the Hypothesis library. These decorators are responsible for generating input data. Line 3 instructs the library to generate random text. Line 4 defines the test itself, which expects an input `s`, which will be randomly generated text as defined on line 3. Finally, line 5 asserts the property that encoding and decoding a certain string results in the original string. This property is checked repeatedly for each of the generated inputs.

Hypothesis, like other PBT tools, supports fine-tuning of the generated test cases. For example, a user can add edge cases that need to be tested in each run alongside the randomly generated inputs by making use of the `@example` decorator. The `@settings` decorator can be used to configure the number of examples to generate or to provide a time budget for the tests. Additionally, a developer can write more complex strategies by making use of decorators such as `@composite` or `@builds` to create data generation strategies that are specific to each project. This way, data structures such as classes, dictionaries, etc. can be generated.

The Hypothesis library also provides a *health check* to warn about mistakes in tests that may cause difficulties when run. This can include warnings about slow data generation or excessive filtering, so the developer is aware in advance.

III. Methodology

In this study, we investigate how practitioners employ property-based testing in ML projects. To this end, we collect a broad range of open-source ML projects, ranging from data processing frameworks to deep learning or natural language processing libraries. We take into consideration any project that is related to ML: whether it is using a pre-trained ML model, a pipeline for preparing training data, or a library for training the model itself. The problem shared by these projects is that they are difficult to test because they operate on large amounts of data, and that their tests may suffer from the oracle problem.

A. Research Questions

We explore the following research questions related to the use of PBT in ML projects:

- **RQ1: What is the sentiment of developers on PBT in open-source ML projects?** We manually investigate commit messages and code comments to find the motivation of developers to adopt PBT, and the difficulties they might have experienced in doing so.
- **RQ2: Which data generation strategies are used in the property-based tests?** Because ML projects often deal with difficult or large amounts of data, we investigate how developers implement data generation strategies, and whether they use external libraries to this end.
- **RQ3: What part of the project is being tested using PBT?** We look at what parts of the ML project are being tested by the PBTs. We consider as categories under test either the ML (training) code, the trained ML model, or the application logic around it.

B. Data Collection

To answer these research questions, we collect a dataset of software systems that employ at least one ML component and have at least one PBT (making use of the Hypothesis library). While datasets of ML projects on GitHub exist [7], [17], our study focuses on ML projects that use PBT. Therefore, we compose a new dataset that takes this constraint into account. We used the GitHub API to fetch the top-200 projects that have a dependency on the Hypothesis library, sorted by number of stars, to obtain popular GitHub projects that may use PBT. To filter out projects not related to ML, we require each to depend on PyTorch, on Scikit-learn[3], or on Tensorflow[4], which is similar to the strategy followed by Widyasari et al. [17]. We find a total of 58 such projects. The resulting projects may use or train an ML model, or contribute to the data processing pipeline used to feed a deployed model or to train a model under development. Unfortunately, using the GitHub API can lead to false positives. As mentioned by Corgozinho et al. [3], many projects specify a dependency on Hypothesis

[3]https://scikit-learn.org/stable/
[4]https://www.tensorflow.org/

in their GitHub dependencies, but do not use the library in any meaningful manner. To mitigate this threat, we only consider projects that have at least one test file with an import statement from Hypothesis, which reduces the dataset to 32 projects. For the ML aspect of the dataset, we manually inspected each project to see if they indeed have something to do with ML as described above, filtering out another four projects. This brings the total number of projects investigated down to 28. Projects range from having one testing file that uses Hypothesis, to having over 150.

C. Analyzing the Data

To answer **RQ1**, we manually analyze the metadata and code of 58 GitHub projects. We first search through all the files of the latest snapshot of each project to find comments that mention PBT or Hypothesis. Afterwards, we also scan the project's main branch to identify any commits that are related to property-based testing or Hypothesis using the GitHub API. This investigation aims to discover positive and negative opinions of developers towards PBT in ML projects. While not all 58 projects are implementing PBT, all have a dependency on Hypothesis. Therefore, they may have used Hypothesis in the past and since abandoned it, making it worthwhile to investigate. We also investigate whether certain Hypothesis health checks (cf. Section II) have been suppressed, as this can be indicative of developers being aware of issues such as slow data generation.

For **RQ2** and **RQ3**, we manually inspect each of the Hypothesis tests we can identify in the test files and investigate the strategies that they use. To this end, we first look for all files that import Hypothesis, and then search for all decorators indicative of PBTs within these files (such as `@given`). We only consider the dataset of 28 ML projects that have at least one property-based test in their code base. For **RQ2**, we investigate which data generation strategies are employed in the tests: simple strategies that generate a primitive type such as an integer, or composite (making use of the `@composite` decorator) and bespoke strategies of multiple lines of code, or third-party strategies from external libraries for input generation.

RQ3 investigates which parts of the ML project, if any, are tested. We define the following categories:

- *1: A trained ML model is tested.* An ML model that was either trained in the project itself, or a third-party pre-trained model, is present in the tests.
- *2: Code related to training ML models is tested.* Any code related to the training of ML models, such as the definition of a loss function.
- *3: Code that is unrelated to any ML model.* The ML projects in our dataset may also feature application logic independent of an ML model.

We make this distinction as property-based tests for code in category 1 and 2 may face ML-specific issues such as the oracle problem and having to generate a large amount of diverse data. Property-based tests for code in category 3 do not face these issues.

To answer this research question, we manually inspect all test files that contain a Hypothesis import to understand what is being tested. In cases where PBTs are easily identifiable, we examine the data generated by the strategies (e.g., whether input data for an ML model is generated) and the functions that are being tested. This entails opening each of the test files that contain a Hypothesis import and inspecting each of the PBTs to categorize the code being tested. Some understanding of the code is required to get an idea of what is being tested. For this preliminary study, we only take into consideration PBTs that are identifiable and understandable.

IV. Results

This section discusses the results of the three research questions defined in Section III. The collected dataset is also available[5].

A. RQ1: Opinions on PBT in open-source ML projects

In order to uncover issues that may arise in PBT, we investigated the commit messages and code comments of 58 ML projects. We also took into account the projects that have a dependency on Hypothesis, but do not use it in their test file, to investigate the reasons why developers may adopt or abandon PBT in their ML projects. Unfortunately, only the projects that still contain at least one import from Hypothesis had notes related to it, leaving us with no insight in this regard.

We manually classify comments as negative, positive, or neutral. Additionally, we also included the suppression of the Hypothesis health check. These are classified as neutral in sentiment, as they indicate that developers are aware of a flagged problem but do not find it problematic. The results are given in Table I. We report for each sentiment in how many projects it is expressed in either the code base or a commit message. Because some items are reported in both commits and comments, we also provide a column "unique" with the total number of projects that mention any sentiment regarding PBT in either the code or the comments.

In total, 16 unique projects mention Hypothesis or PBT in either their commit messages or source code. Of those 16, 14 had notes that can be categorized as negative, whereas six also included positive comments or commits. One of the most common notes, mentioned in eight unique projects, involves having to adjust the time budget for these tests, or having issues with the timeout (e.g., the data generation strategy taking too long). Five projects also mention tests being slow. This can also be seen in the source code itself: at least five of the 28 projects that use Hypothesis suppress the health check that checks for slowness of the data generation.

The random data generation can lead to flakiness (where tests intermittently fail or pass), as mentioned in seven unique projects. The issues related to CI are often also related to tests being either slow or flaky. The Hypothesis health check itself can also lead to flakiness, as reported on by some commit messages.

[5]https://doi.org/10.5281/zenodo.13341915

		in code	in commits	unique
negative	deadline or timeouts	2	6	8
	slowness	5	3	5
	memory consumption	1	0	1
	flaky tests	3	5	7
	issues related to CI	2	5	6
neutral	health check: too slow	5	2	5
	health check: data too large	2	2	3
	health check: filter too much	3	3	3
positive	usefulness data generation	2	3	5
	intuitive and simple	1	0	1
	test case reduction	1	0	1
	broader test coverage	3	1	3

TABLE I
WAYS IN WHICH PBT IS MENTIONED BY GITHUB ML PROJECTS.

		Number of projects
self-written	Only simple strategies	5
	One or more complex strategies	20
uses libraries	`Pandera`	1
	`Caffe2`	1
	`Hypothesis-bio`	1
	`Polars`	1
	`torch-hypothesis`	1
	`hypothesis.extra.pandas`	1
	`hypothesis.extra.numpy`	9
	`hypothesis.extra.tzst`	1
	`hypothesis.extra.redis`	1

TABLE II
HOW PROJECTS IMPLEMENT STRATEGIES AND USED LIBRARIES.

Six projects have positive notes related to PBT or Hypothesis. Two projects include a file that expresses a positive sentiment on data generation, and three other projects mention it in commit messages, making for a total of five unique projects. Three projects also mention the benefits of a broader test coverage specifically. Additionally, one project also mentions PBT being intuitive and simple, and provides a basic tutorial on how to write PBTs for contributors. The developers especially like the test case reduction that Hypothesis provides.

RQ1 In total, we identified negative sentiments related to PBT in 14 projects, whereas six projects feature a positive sentiment. The most common positive sentiment on PBT in ML projects is random data generation for test cases being useful. A recurring negative sentiment is data generation being too slow and it resulting in flaky tests, especially in CI.

B. RQ2: Data generation strategies

In RQ2, we study the data generation strategies employed by developers in ML projects. For projects where developers (partially) implement their own data generation strategies, we classify these as either using *simple* or *complex* strategies. We do so by looking at all the strategies within a project. If all strategies are *simple*, we classify the project as using simple strategies. If at least one *complex* strategy is present, we classify the project as such. Additionally, we also investigate which libraries are used for data generation, if any.

We categorize strategies as being *simple* if they consist of less than ten lines of code and only generate primitive types, such as integers or booleans. We categorize strategies as *complex* if they are tailored to one project or test case. Complex strategies can make use of decorators such as `@composite` or `@builds` to create complex data structures. We also categorize strategies that use only the `@given` decorator but generate data for more than ten different parameters as complex. Complex strategies may also be used by multiple tests across the same projects by defining it as a function.

The results of our investigation are shown in Table II. Only five projects used simple generation strategies. These five projects were also projects that had, on average, fewer PBTs in comparison to projects that implemented more complex data generation strategies. Most projects employ complex data generation strategies, with 20 projects using decorators to make composite strategies, having test cases that generate multiple parameters simultaneously, etc. In comparison, Corgozinho et al. [3] found that 65% of PBTs used Hypothesis' strategies. However, the study investigated individual PBTs in a broad range of projects, whereas we investigated all PBTs within ML projects. The remaining 3 projects of the 28 studied for this research question only used external libraries.

We also look into the use of external libraries and Hypothesis' first-party extension. 13 different projects either use at least one external library or the Hypothesis extension. In rare cases, developers used uncommon libraries to aid with the generation of complex, domain-specific data structures. For instance, to generate biological data formats such as protein structures or to generate PyTorch-related structures.

Hypothesis also provides some first-party extensions, prefixed in Table II with `hypothesis.extra`, to simplify data generation for certain libraries. The most commonly used is NumPy, used by nine out of 28 projects. NumPy itself is also a popular library among ML projects.

RQ2 20 out of 28 projects use at least one self-written, complex data generation strategy. This may be attributed to the fact that ML projects must often process large amounts of data, and that it is difficult to specify how this data should be automatically generated. Hypothesis' first-party extensions are also used by multiple projects, with the NumPy extension used in 9 out of the 28 ML projects. External libraries can fill a gap in defining strategies that generate domain-specific data structures.

C. RQ3: Parts of the ML project being tested

In RQ3, we investigate which parts of the ML project are tested. As mentioned in Section III-C, we inspect the PBTs in each of the 28 ML projects and assess whether they test a (pre-)trained ML model, test ML-related code, or test code that is unrelated to ML. The results of this research question are shown in Table III. For some tests, the purpose of the test was not immediately clear to an outside observer (marked as

Tests ...	trained model	ML training code	unrelated code
Yes	3	13	20
No	20	14	6
Ambiguous	5	1	2

TABLE III
HOW OFTEN DIFFERENT PARTS OF THE ML PROJECT ARE TESTED.

ambiguous in the table). Some projects had, for example, clear PBTs related to testing ML code, but in terms of testing trained models it was more ambiguous. Therefore, some projects can be ambiguous in one category but not in another.

Few projects use (pre-)trained models in their property-based tests, with only three projects we could identify. For five projects, it is ambiguous. 13 projects test code related to training ML models. It is important to note, however, that some ML projects in our dataset do not necessarily have code that is related to training an ML model. As mentioned in Section III, we consider all projects that are related to ML. Some of the projects, for example, are data processing frameworks that can be used together with other ML libraries, but do not train ML models themselves. MLOps tools are another example of this. In total, at least seven of the 28 projects in the dataset do not train an ML model itself, with another four being undecided. Projects that do not train an ML model will therefore also not be able to test ML training code using PBT. 20 of the 28 projects also test code unrelated to the ML model, with two projects where it remains ambiguous. Developers hence not only use PBT to test their ML code, but also conventional application logic.

RQ3 Only three projects use PBT together with a pre-trained model. Using PBT for testing ML training code is more common. Using PBT for conventional code is the most common. This code is also less likely to deal with some of the issues related to testing ML, such as the oracle problem. We discuss other possible reasons for this in Section VII.

V. THREATS TO VALIDITY

For this work, we created a dataset of 58 projects. Of these, only 28 were usable for RQ2 and RQ3. As we selected projects based on stars, only popular projects were included, which might not be representative of all ML projects. This work also only considers Python projects that use the Hypothesis library. However, Python is one of the most common languages in ML projects, and Hypothesis is the most popular PBT framework for Python. Additionally, the manual labelling in this work was performed by one author, which can lead to errors or bias in the classification.

VI. RELATED WORK

There have been studies that explore porting PBT to ML projects (e.g. [11], [12], [14], [18]). These studies either describe how developers can use PBT within ML projects (by, for example, providing a set of possible properties), or explore novel ways of performing property-based testing within ML projects. However, in our study, we look at how developers are performing PBT in ML projects. To our knowledge there are no other studies exploring this.

Previous work by Corgozinho et al. [3] looked into 86 property-based tests written in Hypothesis, across 30 popular GitHub repositories. It examined how developers implement these tests, and what features or decorators of Hypothesis are most used. The study divides PBTs into 10 categories based on the structure of the test itself. Our study differs by targetting ML projects in particular and by investigating the difficulties developers experience when implementing these tests. Furthermore, our study goes beyond the structure of these tests by looking into the data generation strategies that are employed and into which parts of the project are tested.

A preliminary qualitative study by Goldstein et al. [6], and followed by [5], looked into how property-based testing is used in an industrial (non-ML) setting by interviewing developers of one company that makes use of property-based testing. Similarly, Arts et al. [1] performed a case study on using PBT in early development in an industrial setting. Our empirical study considers a larger variety of open-source projects. It also focused on ML projects in particular, to identify issues faced by developers that use PBT in ML projects.

VII. CONCLUSION AND FUTURE WORK

In this preliminary study, we investigated the use of PBT in 58 open-source ML projects, as PBT of ML projects faces some unique challenges. Examples include the oracle problem (ML lacking a clear test oracle and a repetition of the training phase being able to lead to a different behaviour [13]) and having to deal with a larger quantity of data. We find that issues commonly reported by developers of these projects include slow data generation, having to change time budgets for running the tests, and test flakiness. At the same time, some projects mention the usefulness of data generation. This automated data generation allows for broader test coverage.

In terms of the implemented data generation strategies, 20 out of 28 projects define at least one or more complex strategies. This can point to the fact that it is indeed more difficult to specify how automated data generation should be performed for testing ML projects.

Few of the studied projects use PBT to test a (pre-)trained ML model. However, testing code related to ML training is more common. Tests for conventional code were the most common across the studied projects. We hypothesize that reasons for this discrepancy may include the test oracle problem and the fact that ML code is often written by data scientists, whereas the rest of the software might be written by software engineers who have a different skillset and knowledge of different frameworks. In future research, we intend to verify this hypothesis in a larger study that takes into consideration the different types of ML projects.

VIII. ACKNOWLEDGEMENT

This work was supported by Research Foundation – Flanders (FWO) (grant number 1SHFI24N).

REFERENCES

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10, 2006.

[2] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[3] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. How developers implement property-based tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 380–384. IEEE, 2023.

[4] Michael Felderer, Barbara Russo, and Florian Auer. On testing data-intensive software systems. *Security and Quality in Cyber-Physical Systems Engineering: With Forewords by Robert M. Lee and Tom Gilb*, pages 129–148, 2019.

[5] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[6] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some problems with properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*, 2022.

[7] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International conference on mining software repositories*, pages 431–442, 2020.

[8] JetBrains. Python developers survey 2022 results. https://lp.jetbrains.com/python-developers-survey-2022/.

[9] David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[10] Alina Mailach and Norbert Siegmund. Socio-technical anti-patterns in building ml-enabled software: insights from leaders on the forefront. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 690–702. IEEE, 2023.

[11] Christian Murphy, Gail E Kaiser, and Lifeng Hu. Properties of machine learning applications for use in metamorphic testing. 2008.

[12] Mohammad Rezaalipour and Carlo A Furia. An annotation-based approach for finding bugs in neural network programs. *Journal of Systems and Software*, 201:111669, 2023.

[13] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25:5193–5254, 2020.

[14] Arnab Sharma, Caglar Demir, Axel-Cyrille Ngonga Ngomo, and Heike Wehrheim. Mlcheck–property-driven testing of machine learning classifiers. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 738–745. IEEE, 2021.

[15] Arnab Sharma, Caglar Demir, Axel-Cyrille Ngonga Ngomo, and Heike Wehrheim. Mlcheck-property-driven testing of machine learning models. *arXiv preprint arXiv:2105.00741*, 2021.

[16] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[17] Ratnadira Widyasari, Zhou Yang, Ferdian Thung, Sheng Qin Sim, Fiona Wee, Camellia Lok, Jack Phan, Haodi Qi, Constance Tan, Qijin Tay, et al. Niche: A curated dataset of engineered machine learning projects in python. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 62–66. IEEE, 2023.

[18] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.