What Are We Really Testing in Mutation Testing for Machine Learning? A Critical Reflection

Panichella, A.; Liem, C.C.S.

DOI
10.1109/ICSE-NIER52604.2021.00022

Publication date
2021

Document Version
Final published version

Published in
43rd International Conference on Software Engineering - New Ideas and Emerging Results

2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 978-1-6654-0140-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-NIER52604.2021.00022

What Are We Really Testing in Mutation Testing for Machine Learning? A Critical Reflection

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Cynthia C. S. Liem
c.c.s.liem@tudelft.nl
Delft University of Technology
Delft, The Netherlands

***Abstract*—Mutation testing is a well-established technique for assessing a test suite's quality by injecting artificial faults into production code. In recent years, mutation testing has been extended to machine learning (ML) systems, and deep learning (DL) in particular; researchers have proposed approaches, tools, and statistically sound heuristics to determine whether mutants in DL systems are killed or not. However, as we will argue in this work, questions can be raised to what extent currently used mutation testing techniques in DL are actually in line with the classical interpretation of mutation testing. We observe that ML model development resembles a test-driven development (TDD) process, in which a training algorithm ('programmer') generates a model (program) that fits the data points (test data) to labels (implicit assertions), up to a certain threshold. However, considering proposed mutation testing techniques for ML systems under this TDD metaphor, in current approaches, the distinction between production and test code is blurry, and the realism of mutation operators can be challenged. We also consider the fundamental hypotheses underlying classical mutation testing: the competent programmer hypothesis and coupling effect hypothesis. As we will illustrate, these hypotheses do not trivially translate to ML system development, and more conscious and explicit scoping and concept mapping will be needed to truly draw parallels. Based on our observations, we propose several action points for better alignment of mutation testing techniques for ML with paradigms and vocabularies of classical mutation testing.**

***Index Terms*—mutation testing, machine learning, mutation operators, software testing**

I. Introduction

In many present-day systems, automated data processing powered by machine learning (ML) techniques has become an essential component. ML techniques have been important for reaching scale and efficiency in making data-driven predictions which previously required human judgment. In several situations, ML models—especially those based on Deep Learning (DL) techniques—even have been claimed to perform 'better than humans' [1], [2]. At the same time, this latter claim has met with controversy, as seemingly well-performing ML models were found to make unexpected mistakes that humans would not make [3], [4].

One could argue that ML procedures effectively are software procedures, and thus, that undesired ML pipeline behavior signals faults or bugs. Therefore, a considerable body of 'testing for ML' work has emerged in the software engineering community, building upon established software testing techniques, as extensively surveyed by Ben Braiek & Khom [5], as well as Zhang et al. [6]. While this is a useful development, in this work, we argue that *current testing for ML approaches are not sufficiently explicit about what is being tested exactly*. In particular, in this paper, we focus on mutation testing, a well-established white-box testing technique, which recently has been applied in the context of DL. As we will show, several fundaments of classical mutation testing are currently unclearly or illogically mapped to the ML context. After illustrating this, we will conclude this paper with several concrete action points for improvement.

II. ML Model Development

When a (supervised) ML model is being developed, first of all, a domain expert or data scientist will collect a dataset, in which each data point evidences input-output patterns that the ML model should learn to pick up. When input from this dataset is offered to a trained ML model, the model should be capable of yielding the corresponding output.

In contrast to the development of traditional software [5], [6], the process that yields the trained model will partially be generated through automated optimization routines, for which implementations in ML frameworks are usually employed. The data scientist explicitly specifies the type of ML model that should be trained, with any hyperparameters of interest. This choice dictates the architecture of the model to be constructed, and the rules according to which the model will be iteratively fitted to the data during the training phase.

Considering that the training data specifies what output should be predicted for what input, and the goal of model fitting is to match this as well as possible, we argue that *the ML training procedure resembles a test-driven development (TDD) procedure*. Under this perspective, *the training data can be considered as a test suite*, which is defined before system development is initiated. Initially, as training starts, a model M will not yet fit the data well; in other words, too many tests will fail. Therefore, a new, revised version M' of the model will be created. The more tests will be passing, the closer the model will be considered to be to 'the correct program'. The process is repeated until convergence, and/or until a user-defined performance threshold will be met.

As soon as model training is finished, the performance of the trained model will be assessed against a test set that

978-1-6654-0140-1/20/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-NIER52604.2021.00022

was not used during training. As a consequence, performance evaluation of a trained model can be seen as *another TDD procedure*, now *considering the test data as the test suite*. If too many test cases fail, the trained model is unsatisfactory, and will need to be replaced by another, stronger model. Here, the 'other, stronger model' can be considered in a black-box fashion; it may be a variant of the previously fitted model, or a completely different model altogether.

Finally, as soon as the satisfactory performance is reached, a trained model will likely be integrated as a component into a larger system. While this integration is outside the scope of the current paper, we would like to note that practical concerns regarding ML faults often are being signaled in this integrated context (e.g. unsafe behavior of a self-driving car), but consequently may not only be due to the ML model.

III. Classical Mutation Testing

In 'traditional' software testing, mutation testing is an extensively surveyed [7]–[9], well-established field, for which early work can be traced back to the 1970s.

The process. Given a program P, mutation testing tools generate program variants (i.e., *mutants*) based on a given set of transformation rules (i.e., *mutation operators*). The mutants correspond to potential (small) mistakes that *competent programmers* could make and that are artificially inserted in the production code. If the test suite cannot detect these injected mutants, it may not be effective in discovering real defects. More precisely, an effective test case should pass when executed against the original program P, but fail against a mutant P'. In other words, the test should differentiate between the original program and its mutated variant. In this scenario, the mutant P' is said to be *killed*. Instead, the mutant P' *survives* if the test cases pass on both P and P'. The test suite's effectiveness is measured as the ratio between the number of killed mutants and the total number of non-equivalent mutants being generated (*mutation score*) [7]. Therefore, changes (mutants) are applied to the production code, while the test code is left unchanged. Mutants can only be killed if they relate to tests that passed for the original program. After mutation analysis, test code can be further improved by adding/modifying tests and assertions.

The hypotheses. As reported by Jia and Harman [7], mutation testing targets faults in programs that are close to their correct version. The connection between (small) mutants and real faults is supported by two fundamental hypotheses: the *Competent Programmer Hypothesis* (CPH) [10] and the *Coupling Effect Hypothesis* (CEH) [11]. Based on these hypotheses, mutation testing only applies simple syntactic changes/mutants, as they correspond to mistakes that a competent programmer tends to make. Furthermore, each (first-order) mutant P' includes one single syntactic change, since a test case that detects P' will also be able to detect more complex (higher-order) faults coupled to P'.

Challenges in mutation testing: Prior studies [12]–[14] showed that mutants are valid substitutes of real faults and, therefore, mutation testing can be used to assess the quality of a test suite. However, mutation testing has some important challenges: the high computation cost and equivalent mutants. *Equivalent mutants* are mutants that always produce the same output as the original program; therefore, they cannot be killed [7]. While *program equivalence* is an undecidable problem, researches have proposed various strategies to avoid the generation of equivalent mutants for classical programs [7].

IV. Mutation Testing Approaches for ML

Mutation testing has also been applied to ML software. To demonstrate the effectiveness of metamorphic testing strategies for ML classification algorithm implementations, in Xie et al. [15], mutation analysis was performed using classical syntactical operators. With the recent rise in popularity of deep learning (DL) techniques, accompanied by concerns about DL robustness, interest in mutation testing has re-emerged as a way to assess the adequacy of test data for deep models. To this end, in 2018, two mutation testing proposals were proposed in parallel: MuNN by Shen et al. [16], and DeepMutation by Ma et al. [17], both seeking to propose mutation operators tailored to deep models. Both methods focus on convolutional neural network (CNN) models for 'classical' image recognition tasks (MNIST digit recognition in MuNN and DeepMutation, plus CIFAR-10 object recognition in DeepMutation).

MuNN considers an already trained convolutional neural network as production code, and proposes 5 mutation operators on this code, that consider deletions or replacements of neurons, activation functions, bias and weight values. DeepMutation argues that the 'source of faults' in DL-based systems can be caused by faults in data and the training program used for model fitting. As such, the authors propose several *source-level mutation operators*, which generate mutants *before* the actual training procedure is executed. In their turn, the source-level operators are divided into *operators that affect training data* (e.g. Label Error, Data Missing) and *operators that affect model structure* (e.g. Layer Removal). Besides this, the authors also propose eight *model-level mutation operators*, that operate on weights, neurons, or layers of already-trained models. These model-level operators can be compared to the operators in MuNN; indeed, one can e.g. argue that DeepMutation's Gaussian Fuzzing operator is a way to implement MuNN's Change Weight Value operator, where in both cases, weights in the trained model will be changed.

Noticing that DL training procedures include stochastic components, such that re-training under the same conditions will not yield identical models, and that hyperparameters may affect the effectiveness of mutation operators, Jahangirova & Tonella revisit the operators of MuNN and DeepMutation from a stochastic perspective [18] and study operator (in)effectiveness, also adding a regression task (steering wheel angle prediction from self-driving car images) to the MNIST and CIFAR-10 classification tasks. Furthermore, the authors study more traditional syntactic mutations on the training program, employing the `MutPy` tool. Similarly, Chetouane et al. [19] do this for neural network implementations in

```
1  @Test
2  public void testLearning() {
3     List<Image> trainingData =
           FileUtils.readDataset('training_data');
4     Model m = Model.train(trainingData);
5     double counter = 0;
6     for (Image image : trainingData){
7        String outcome = m.predict(image);
8        String expected = image.getGroundTruth();
9        if (outcome.equals(expected))
10          counter++;
11    }
12    double accuracy = counter / trainingData.size();
13    assertTrue(accuracy >= 0.9);
14 }
15
16 @Test
17 public void testPerformance(){
18    List<Image> testData =
           FileUtils.readDataset('test_data');
19    Model m = Model.readModel('model_path');
20    double counter = 0;
21    for (Image image : testData){
22       String outcome = m.predict(image);
23       String expected = image.getGroundTruth();
24       if (outcome.equals(expected))
25          counter++;
26    }
27    double accuracy = counter / testData.size();
28    assertTrue(accuracy >= 0.9);
29 }
```

Listing 1. Tests for ML training and performance assessment

Java, employing the `PIT` tool [20]. Finally, seeking to expand mutation testing beyond CNN architectures, the DeepMutation++ tool [21] both includes DeepMutation's CNN-specific mutation operators and various operators for Recurrent Neural Network architectures.

V. Was Mutation Testing Truly Applied to ML?

A. Is the production code being mutated?

Reconsidering our framing in Section II of ML model development as two TDD procedures (one focusing on model fitting, and one focusing on performance evaluation of a trained model), this leads to two types of production code, with two corresponding (implicit) test suites: respectively, the data referred to as 'training data' and 'test data' in ML vocabulary.

We can frame the assessments of how well a model has been trained, and how well it performs after training, as separate test cases. Listing 1 illustrates this for a fictional image recognition task implemented in Java. The `testLearning` method assesses the success of a training procedure: it reads the training set (line 3), runs the training procedure for a given model (line 4), and lets the test pass in case a user-provided success criterion (here: a minimum accuracy threshold) is met (line 14). The `testPerformance` method runs predictions by an already-trained model on a given dataset, again letting the test pass in case the user-provided success criterion is met.

Under this framing, we question to what extent the operators proposed for DL testing (Section IV) are true mutation operators. In fact, we argue that the category of post-training model-level mutation operators is the only one clearly fitting the mutation testing paradigm, in which production code is altered to assess the quality of the test suite—while still raising questions regarding the fit to the fundamental mutation testing hypotheses, as we will explain in Section V-B.

Many source-level mutation operators actually target the model's (so the production code's) correctness, rather than the quality of the test suite. For instance, let us consider the *Label Error* operator, which changes the label for one (or more) data point(s) in the training set. Considering the TDD metaphor for ML training, this operator alters the test suite (training data), rather than the production code that generates or represents the trained model. Furthermore, the changes are applied to assess that the model trained on the original dataset provides statistically better predictions than a model[1] trained with faulty data labels. This therefore appears to be test amplification rather than a mutation testing, and this consideration holds for any proposed data-level operator.

For source-level model structure operators, the situation is ambiguous. The code written by the data scientist to initiate model training may either be seen as production code, or as 'the specification that will lead to the true production code' (the trained model). It also should be noted that this 'true production code' is yielded by yet another software program implementing the training procedure, that typically is hidden in the ML framework.

B. Revisiting the fundamental mutation testing hypotheses

We argue that the two fundamental hypotheses of mutation testing should be reconsidered for ML: many terms in these hypotheses are not easily mapped to ML contexts.

> **CPH** Competent programmers "tend to develop programs close to the correct version." [10]

First of all, *for ML, not all 'programmers' are human, and the degree of their competence may not always show in production code.* As discussed in Sections II and IV, when an ML model is to be trained, the general setup for the training procedure will be coded by a human being (often, a data scientist). This program could be considered as human-written production code, but is not the true production code: it triggers an automated optimization procedure (typically previously authored by an ML framework programmer), which will yield the target program (the trained model).

Questions can be raised about *what makes for 'realistic' mistakes*. Currently proposed operators on data (which, as we argued in Section V-A, should be interpreted as operators for test amplification, rather than for mutation testing) largely tackle more mechanical potential processing issues (e.g., inconsistent data traversal and noisy measurement), rather than faults that a human operator would make. Generally, the more interesting faults made during dataset curation can have considerable effects downstream, but are not necessarily encoded in the form of small mistakes. For example, if biased data is fed to the pipeline, this may lead to discriminatory decisions downstream. However, one cannot tell from a single data point whether it resulted from biased curation. Furthermore, the undesired downstream behavior will also not be trivially

[1]Same model, with same training algorithm, and same parameter setting.

traceable to source code; instead, it rather is a signal of implicit, non-functional requirements not having been met. Thus, higher-level acceptance testing will be needed for problems like this, rather than lower-level mutation testing strategies.

As mentioned in Section V-A, the post-training model-level operators proposed in literature fit the classical mutation testing paradigm. Yet, some of them are unlikely to represent realistic faults made by the 'programmer' (the ML training framework). The way the framework iteratively optimizes the model fit is bound to strict and consistent procedures; thus, it e.g. will not happen that subsequent model fitting iterations will suddenly change activation functions or duplicate layers.

Finally, *the concept of 'an ML system close to the correct version' is ill-defined.* In contrast to classical software testing setups, where all tests should pass for a program to be considered correct, this target is fuzzier in ML systems. During the training phase, one wants to achieve a good model fit, but explicitly wish to *avoid* a 100% fit on the training data, since this implies overfitting. As soon as a model has been trained, performance on unseen test data is commonly considered as a metric that can be optimized for. Still, the test data points may not constitute a perfect oracle. As a consequence, perfect performance on the unseen test set may still not guarantee that an ML pipeline behaves as intended from a human, semantic perspective [22], [23]. Thus, when considering mutation testing for ML, we actually cannot tell the extent to which the program to mutate is already close to the behavior we ultimately intend.

> **CEH** "Complex mutants are coupled to simple mutants in such a way that a test dataset that detects all simple mutants in a program will also detect a large percentage of the complex mutant." [11]

In ML, we still lack a clear sense of what makes for simple or complex mutants. As noted in [18], the stochastic nature of training may lead to multiple re-trainings of the same model on the same data leading to models that may differ in weights and output. Also, when considering mutations after a model has been trained, not all neurons and layers have equal roles, so the effect of mutating a single neuron may greatly differ.

VI. Conclusion and future work

In this paper, we discussed how classical hypotheses and techniques for mutation testing do not always clearly translate to how mutation testing has been applied to DL systems. We propose several action points for SE researchers to consider, when seeking to further develop mutation testing for ML systems in general, while staying aligned to the well-established paradigms and vocabularies of classical mutation testing.

When defining new mutation operators for ML, be explicit about what production and test code are. In classical mutation testing, there is a clear distinction between the production code (to be altered by mutation operators) and the test code (to be assessed by mutation testing). This clear distinction must be kept when applying mutation testing to ML systems.

Clearly settle on what the system under test (SUT) is when extending software testing techniques to ML systems. Currently, *mutation testing for DL* is a common umbrella for all techniques that target DL models, the training procedure, and the surrounding software artifacts. However, as we argued in Section II, ML model development (both within and outside of DL) involves several development stages with associated artifacts, each requiring dedicated, different testing strategies. Furthermore, where in traditional ML, one test set suffices for a specific ML task (regardless of what ML model is chosen for this task), the mutation testing paradigm implies that the adequacy of a given test suite may differ, depending on the program it is associated with. Thus, depending on the program, a given test suite may need different amplifications.

Work with experts to better determine what 'realistic mutants', 'realistic faults' and 'realistic tests' are. Ma et al. [17] rightfully argued that not all faults in DL systems are created by humans, but little insight exists yet on how true, realistic faults can be mimicked. Humbatova et al. [24] recently proposed a taxonomy of real faults in DL systems, finding that many of these have not been targeted in existing mutation testing approaches yet. Furthermore, when the mutation score will indicate that a given test set is not sufficiently adequate yet, it also still is an open question how test suites can be improved: test cases need to both meet system demands, while having to be encoded in the form of realistic data points. Here, insights from domain and ML experts will be beneficial.

Investigate what 'a small mutant' is. This aspect is rather complex in ML: repeating the exact same training process on the exact same data is not mutation, but may yield major model differences, due to stochasticity in training. Similarly, a small back-propagation step may affect many weights in a DL model. At the same time, given the high dimensionality of data and high amounts of parameters in DL models, other mutations (e.g. noise additions to pixels) may be unnoticeable as first-order mutants, and may need to be applied in many places at once to have any effect.

Finally, we wish to point out that *the evolution of ML programs differs from 'traditional' software*. In traditional software, revisions are performed iteratively, applying small improvements to previous production code. This is uncommon in ML, where revisions are typically considered at the task level (e.g. 'find an improved object recognition model'), rather than at the level of production code (e.g. 'see whether changing neuron connections in a previously-trained deep neural network can improve performance'). Translated to software engineering processes, this is more similar to re-implementing the project many times by different, independent developer teams, than to traditional software development workflows. In such cases, if the production code of trained models will not be expected to evolve, employing black-box testing strategies may overall be more sensible than employing white-box testing strategies. Again, this question requires collaboration and further alignment between software engineering, machine learning, and domain experts, which may open many more interesting research directions.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[3] B. L. Sturm, "A Simple Method to Determine if a Music Information Retrieval System is a "Horse"," *IEEE Transactions on Multimedia (TMM)*, vol. 16, no. 6, pp. 1636–1644, Oct 2014.

[4] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[5] H. Ben Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, 2020.

[6] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," *IEEE Transactions on Software Engineering (TSE)*, 2020.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 5, pp. 649–678, 2011.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[9] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," *Software Testing, Verification and Reliability (STVR)*, vol. 28, no. 6, p. e1675, 2018.

[10] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[12] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[13] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 537–548.

[14] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th ACM/IEEE International Conference on Software Engineering (ICSE)*, 2005, pp. 402–411.

[15] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[16] W. Shen, J. Wan, and Z. Chen, "MuNN: Mutation Analysis of Neural Networks," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, 2018.

[17] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepMutation: Mutation testing of deep learning systems," in *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018.

[18] G. Jahangirova and P. Tonella, "An Empirical Evaluation of Mutation Operators for Deep Learning Systems," in *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[19] N. Chetouane, L. Klampf, and F. Wotawa, "Investigating the Effectiveness of Mutation Testing Tools in the Context of Deep Neural Networks," in *Proceedings of the 16th International Work-Conference on Artificial Neural Networks*, 2019.

[20] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A Practical Mutation Testing Tool for Java (Demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, p. 449–452.

[21] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "DeepMutation++: a Mutation Testing Framework for Deep Learning Systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[22] L. Bertinetto, R. Mueller, K. Tertikas, S. Samangooei, and N. A. Lord, "Making Better Mistakes: Leveraging Class Hierarchies with Deep Networks," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[23] C. C. S. Liem and A. Panichella, "Oracle Issues in Machine Learning and Where To Find Them," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020.

[24] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020.