A Framework for Mutation Testing of Machine Learning Systems

Raju Singh
Department of Computer Science & Information Systems
BITS Pilani
Pilani, India
p20200106@pilani.bits-pilani.ac.in

Mukesh Kumar Rohil
Department of Computer Science & Information Systems
BITS Pilani
Pilani, India
rohil@pilani.bits-pilani.ac.in

Abstract—**In this paper, we provide an insight journey of Testing of Machine Learning Systems (MLS), its evolution, current paradigm, and we propose a machine learning mutation testing framework with scope for future work. Machine Learning (ML) Models are being used even in critical applications such as Healthcare, Automobile, Air Traffic control, Share Trading, etc., and failure of an ML Model can lead to severe consequences in terms of loss of life or property. To remediate this, the ML community around the world, must build highly reliable test architectures for critical ML applications. At the very foundation layer, any test model must satisfy the core testing attributes such as test properties and its components. These attributes should come from the software engineering discipline but the same cannot be applied in as-is form to the ML testing and in this paper, we explain why it is challenging to use Software Engineering Principles as-is when testing any MLS.**

Keywords-Machine Learning, Software Testing, Quality Attributes, Deep Learning, Model Mutation Testing.

I. INTRODUCTION

In the current context of software development and machine learning (ML), it is inevitable, not to come across an ML scenario in day to day life. It spans across business critical applications such as share trading, insurance and banking, medical applications such as drug manufacturing, identification of disease and medical imaging, and safety critical applications such as autonomous driving and robotics [1]-[3]. Software testing [4] of mathematical software [5] and Intelligent systems [6] has adopted some of the software testing methodologies. Applications of ML in several critical sectors make ML testing [7] a reliable way to ensure quality and minimize failure scenarios. An adoption of testing framework from traditional software testing with the addition of key ML quality attributes [7] makes more sense. Testing framework that covers performance (for critical real time systems), security (for business applications and health care applications) and safety (for system of systems) increases its trustworthiness.

To better understand the testing challenges for ML systems [8]-[10], we need to deep dive as how ML systems are different from traditional software system. Traditional software is more deterministic in nature, lacks dynamicity in terms of varied inputs. On the other hand, ML systems are dynamic, non-deterministic and expected to learn from data (labels), and predict the output accordingly.

For instance, a rover has to determine the path on a rocky terrain based on the imaging data that it gathers from the surrounding, the forest fire alert systems have to generate a prediction based on the environmental data such as air humidity, wind, temperature and climatic conditions. The model tends to evolve and learn from historical data.

Oracle Problem [11]: Machine learning models are difficult to test because they are designed to solve problem based on learning from past experience (label data, supervised learning), without past experience (unsupervised learning) or through re-enforcement. Attempts have been made to draw parallel between the ML testing approach with Software Testing. By understanding the process of software development, we should be able to break down the software stack into components (unifiable units), and build test cases around it. In other approaches, we have Test Driven Development (TDD) to setup in testing framework. This approach might not work well with ML models. It is because, machine learning models are mostly monolith, and components may not reflect the true nature of the ML model as a whole. Again, breaking ML model into unifiable components or developing with TDD is a cumbersome task.

In order to understand and design a test framework for ML system, we need to understand the behaviour [4] of the model, and how the model interacts with the surroundings. Studying behaviour of the ML model, gives limited insight into the model.

Further the paper is organized as: Section II summarises the background and related work, Section III summarizes Machine Learning Testing Scenarios in terms of faults, failures and Oracle problem, Section IV explains machine learning testing by considering dynamicity of MLS, Section V describes proposed framework for machine learning system testing, Section VI describes prediction mutation testing to explore the possibility of testing machine learning models such as Deep Neural Network (DNN) models, Section VII summarizes model mutation testing, Section VIII presents the major challenges, Section IX lists the assumptions and hypothesis for set of transformation rules to yield mutated DNN models, Sections X brings out some approaches for mutation testing, and Section XI furnishes conclusion and scope for future work.

II. BACKGROUND AND RELATED WORK

Mutation testing in traditional software predates any similar testing framework in Machine Learning and DNNs specifically. Here, the mutation testing is a proven tool and has higher

DOI reference number: 10.18293/SEKE2021-155

accuracy. Mutant operators form a well-researched area that has its implementation in several high level programming languages for traditional software. With time and increasing complexity of traditional software, the mutation testing framework has been extended well. Use of mutation testing in machine learning, is being extensively researched and several researchers have established milestones for most of the known DNN models. One such approach is DeepMutation.

Machine Learning: Machine learning is a field of study that gives the ability to computers to learn without being explicitly programmed. A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E [12], [13].

Machine learning is a phased approach. The first phase is the learning phase. In this phase, data is gathered and bucketed as training and test data sets. Training data set is identified by attributes and label. The outcome of this phase is a model that is drawing the relationship between the attributes and the label. The subsequent phase deals with applying the model to different dataset (test data). There are several algorithms to accomplish this, such as classification algorithms, ranking algorithms, etc. There are several attempts and general-purpose availability of model based mutation testing, which depends upon comparing results from different test scenarios.

Terms used in machine learning domain:

Dataset: An ingredient for machine learning model, consists of sets of instances for building or evaluating the model. It is further categorized as:

Training Data: This data is obtained from the sources (sensors, data collection devices, etc. aggregated and cleaned up to exclude bias and noise) and is used for the purpose of training a machine learning model. This model is basically an organic algorithm which learns from the training data and performs a particular task.

Validation data: This data is from the training data, used to tune the hyper-parameters of learning algorithm.

Test data: This data is the part of training data, for which machine learning model has not been trained yet. Based on the performance of the ML model and its behaviour with the test data, we can attribute the machine learning model maturity.

Sub Definitions:

Instance is an information record about the object. *Feature* is a measurable property. Errors are also an important aspect of machine learning, and it is this property that the model behaviour depends on. *Test error* is mainly focused on deviation measure between the obtained value and the expected value.

Let us classify Machine Learning.

Supervised Learning [9]: The goal is to predict the value of an outcome measure based on the number of input measures. It is commonly referred to as regression [14] problem since its outcome measurement is quantitative.

Unsupervised Learning [15]: The goal is to describe the association and patterns in a set of input measures. We only observe the feature and have no measurement about the outcome.

Reinforcement Learning [16]: In this approach, agent (learning system) can observe an environment, selectively perform actions, and get rewards (or penalties). The key here is, it must learn by itself, and accordingly respond in actions for the good. This approach is referred to as policy, to get most reward over a period of time. In nutshell, a policy defines what actions the agent should take when subjected to a condition.

III. Machine Learning Testing Scenarios

Fault and Failures [17], [18]: The following discussion involves classification of the faults and failure scenarios of the ML systems. Since most of the ML systems deal with uncertain components, faults and failures are possible in ML models. These can be handled by creating counter measures to prevent failure scenarios, however, we can have inevitable scenarios.

Definitions in the IEEE Standard Glossary (IEEE 1990) [19]:

Fault: An incorrect step, process, or data definition in a computer program.

Failure: The inability of a system or component to perform its required functions within specified performance requirements.

Data Sensitive Fault: A fault that causes a failure in response to some particular pattern of data.

Program Sensitive Fault: A fault that causes a failure when some particular sequence of program steps is executed.

The core of the testing system is to find the deviation of ML models from the expected outcome.

Oracle: Oracle tests are basically intended towards the Behaviour test. This is a challenging aspect as the behaviour of ML systems is unpredictable, and this unpredictability makes sure sense to build oracle tests. In MLS context, Metamorphic Oracles have gained ground as a feasible approach to infer oracle information from data. Metamorphic oracles insight metamorphic relations between input values, i.e. if a metamorphic relation exists between the inputs, the corresponding MLS outputs must satisfy a pre-existing relation (ideally, equality or equivalence relation). Input data and its dimension poses a greater instability towards the ML testing realm. So, it is vital to choose adequate test data in order to cover impactful dimensions.

IV. Machine Learning Testing

Software testing techniques, such as unit, integration and system testing [20], can be used in ML testing domain. Additionally, in order to address the dynamicity of the ML systems, additional recommendation has been made for ML testing domain which includes input, model and integration testing.

Input Testing: These tests are concentrated on the input data which is used to train the ML model. The core reason of using input test is to minimise the risk of faults. It can be either offline testing or online testing. During the offline testing, it detects faults by alerting the bias in the training data. In online testing, where ML models are expected to predict for unlabelled data, this testing helps on input validation.

Model Testing: Model testing tests the function aspects of the system under test (SUT) (ML model in isolation, without taking any other component into account). It tries to find the faults in the model architecture, training process, etc. It uses accuracy (for classifier) or mean squared error (for regressions [14], [21]). It is sometimes considered as unit testing.

Integration Testing: Integration testing considers the integration aspect of ML models, hardware systems, software systems and their interactions.

System Testing: System testing is a holistic test to evaluate the system's measures under a given requirement.

Black-box and White-box Testing: Black-box testing [22] screens the internal structure of the design, code and its implementation, of ML systems, without having access to the core while white-box testing is crucial as it knows the internal structure of the code, design, implementation and behaviour of the ML systems. This way it makes more sense to use white-box testing in ML model. In contemporary software, source code is the main source of faults or defects. Mutation testing injects modified program code to introduce defects or faults, and this enables the qualitative measurement of test data by detecting manual changes. With the knowledge on such mutation testing framework, we can suggest a Deep Learning (DL) based mutation testing framework with two stage process.

Source level mutation: DL systems depend on the training program and training data. Training process is defined as the articulation of training program on training data. The master source code of training program and master record of the training data is mutated over a period of time during the testing and the deviation of the Model is recorded. The new evolved model, result from the mutation exercise, is set to run through the training set in order to determine the quality of the test data. The mutation operator can be categorized as: data mutation operator and program mutation operator.

Data Mutation Operators: DL model depends heavily on training data. We know that DL model's robustness depends on the underlying data quality. Error introduces at any stage of data collection, data aggregation and data cleaning, and skews the DL model as the data contains noise.

Program Mutation Operators: Training programs in DL systems are coded using high level languages, and use problem specific programming framework. Injection faults in the program would cause unexpected behaviour in the DL systems.

This requires us to carefully craft mutant operations to inject faults into the training program. The kind of fault we can think of now is like, addition and removal of layers from DL models, pass on skewed weights and activation function while training process.

Model Mutation Testing for DL Systems:

Most of the mutation testing frameworks which work efficiently in traditional software systems do not hold ground with the DL models. The problem is, most of the mutation testing from traditional system is written on the source code, or its low level representation such as byte-code. However, model mutation testing can be a better approach and we will show it.

In *source level mutation testing*, the algorithm injects modifications in the training data and training program, while in *model level mutation testing*, the algorithm updates the DL model obtained from the training program. As the expectation remains intact for both approaches, i.e. to evaluate effectiveness and weakness of the test data set, model level mutation testing leads the way forward by directly mutating the DL model.

ML System Attributes:

Security: ML systems are as vulnerable as any other software systems, along with few inherited vulnerability, given the model footprint. Security reciprocates to the robustness.

Efficiency: ML system efficiency reciprocates to accuracy of its prediction.

Fairness: ML systems suffer from statistical problems such as bias, deviations and skewness.

V. Machine Learning Testing Framework

Behaviour Framework: ML system might behave differently given similar data. The main challenge is to identify the extreme boundaries for a given input space. This is similar to boundary-value analysis.

Test Adequacy Criteria: Any test suite woven around an ML system, should satisfy the quality attributes. As the classical approaches (based on the source code control flow [23]) are not relevant to the ML systems, researchers are trying to find out new domain in order to satisfy the test adequacy.

Mutation: In contemporary software testing domain, mutation testing is gaining grounds. It has become an efficient tool to find the faults in the ML systems, by injecting mutants. DeepMutation fundamentally works at the model level, iterates through varying mutation within the boundary space.

DeepMutation: DNNs have gained ground in several critical applications such as healthcare, autonomous vehicle and robotics. Any DNN system can either be a Feedforward Neural Network (FNN) or a Recurrent Neural Network (RNN) system. An FNN system processes the input information at each layer and forwards it to the next layer. This process continues until the decision is reached. This way, the FNN model preserves the local properties of each layer. On the other hand, the RNN extends the Long-Short Term Memory (LSTM) or memory cells and partially propagates the information backward to secure temporal information of sequential inputs. This way, the decision at any stage not only depends on the given input, but also on the current state. This makes RNN reliable for handling sequential data, for instance, Natural Language Processing (NLP). The spirit of mutation in DNN [24], [25] is similar to that of traditional software. The main idea behind DeepMutation is to introduce adequate number of mutants or operators. The mutants must satisfy the quality attributes for the testing framework, such as input (test) data analysis.

Traditional software is built upon decision logic. This logic is implemented in the form of program code, whereas the DL models and systems are guided by the underlying DNN structures and their weights.

The weight of DL system is generally obtained from executing training program on a given training data, and DNN structure is defined as the code of the training program. These are two potential reasons, a deviation in which cause behavioural issue in the DL systems. The mutation operator can be inflicted in either training data set of training program or both. Once the mutant operators are injected, training program is executed on training data to generate mutated DL models.

DeepMutation Testing Framework:

DNN uses high level languages such as Python and R, however DNN is represented as hierarchical data structure. We are going to shortly lay down on to discuss on the mutation testing framework for DL systems. The first step is to design *source level mutation testing operators*. These operators can modify the training data and training program. The basic idea behind this is to improve the data quality evaluation. The fault might be injected manually, or might naturally occur in the training data or in training program. This framework must address the mutated DL models efficiently and address issues such as computation resource requirements, security vulnerability issues. Given this, we must work backward to generate efficient mutant operators. Before we deliver further, we need to elaborate model-level testing.

Model Level Testing: A model is used to represent the desired behaviour of the system under test (SUT), or to represent the testing strategies and a test environment. A model representation of SUT is at abstraction or partial behaviour. We can derive only functional test cases from SUT. The idea here is to come up to the conclusion that how many model level mutation operators would result in the efficient generation of a set of mutations without inducing model level problem.

VI. Prediction Mutation Testing

Prediction Mutation Testing topic tends to attract two sorts of discussion – Mutation Testing approach in Machine learning, and – Machine learning approach in mutation testing. We will talk a little about the latter part, and then resume the discussion on the primary topic which is related to exploring the possibility of testing machine learning models such as DNN using mutation testing, its framework, challenges, pros and cons, future works, etc.

When we discuss the approach of machine learning and its impact on mutation testing, we consider that here mutation testing can be again applied to a software system or machine learning system. However, applying mutation testing in software system is inherently different from applying mutation testing in machine learning, and it is because of the behavioral changes that software systems and machine learning systems exhibit when subjected to mutation testing.

Testing, in general is a powerful and unified (yet distributed) way to evaluate the quality of underlying systems, be it traditional software systems or machine learning systems. In this approach, we tend to generate a large number of mutants and execute against a test suite to check the ratio of killed mutants. This makes mutation testing computationally expensive. So, it is worthwhile to invest some time to learn about predicting behavior of mutation testing. It is important to note here that this approach is based on the classification model. This model predicts whether mutants are killed or survived during the testing without executing it. However, unlike several predicted problems, this approach also suffers from accuracy loss (which we can ignore as it is minimalistic).

In general, mutants are a set of program variants (or training program in machine learning). A set of transformation rules generates mutants from the original program. These mutants are called mutation operator, that seed logic and syntactic changes into the program one at a time.

Killed Mutants: A mutant, killed by a test-suite, if at least of the test from the source has a varied execution behavior on the mutants and original program. Such mutants are called *killed mutants*. Elsewise, the mutant is known to have survived. The ratio of killed mutants to all mutants (non-equivalents) is referred as mutation score. It is usually used to evaluate the test suite's effectiveness.

Other areas, where use of mutation testing is prevalent are simulation testing, localizing faults, model transformation and guided test generation.

As mentioned earlier, mutation testing is an extremely expensive approach. It requires generating and executing each mutant on the test suite. Both of these activities – generation of mutations and execution of mutants, are expensive operations on hardware of scale. In recent times, however, we have seen phenomenal progress to bring down the operational cost for mutant generation, though executions remain expensive in spite of several refinement techniques such as selective mutation testing, weak mutation testing, high-order mutation testing, optimized-mutation testing, etc.

Due to the problems faced for the expense versus effectiveness, predictive mutation testing is gaining ground. Mutation testing enables machine learning to build a predictive model by means of collecting a series of features. These features can be test-suite coverage or mutation operators on already executed mutants of earlier versions of the project or even other projects.

Earlier versions of same projects are commonly referred to as cross-version prediction. Cross project predictions are referred to other projects under test.

Tradeoff - Efficiency versus Effectiveness:

Any prediction model inherently suffers from accuracy problem. However, several experimental approaches in predication mutation testing domain have shown positive signs as it improves efficiency and accuracy of mutation testing. This is a clear indication of how prediction mutation testing stands out of traditional mutation testing. It will be worthwhile to look into the class probability distribution provided by the classifier, with which developer may choose the mutant with proper probability distribution in order to get better prediction result. It is a considerable improvement over traditional mutation testing, as it is light weight, in-expensive comparatively, with relatively high accuracy. In this article, it is assumed that mutation testing refers to prediction mutation testing.

VII. Model Mutation Testing

Models are common in software testing. It is used to select test suites. Applications of mutation testing at a model level can contribute to reliable and early assessment of the quality of the test suits. This can also help in defining a test suite which has high fault detection rates. One of the issues which we observed while using mutation testing [18] at the early development stage, is related to its reliability and quantifying it.

VIII. Challenges

Contemporary coding for functional requirements is different from programming a DL model. The basic difference is, in contemporary programming, we break down the monolith requirements into small chunks of programmable units. Each unit is programmed separately and satisfies the software quality attributes such as correctness, fairness, security, etc. and then the units can be combined together with other modules to form the holistic program. In this approach, each unit or module has its own logic, an aggregation of which comply with the integrity attributes of the whole program.

While in DL systems, which are fundamentally data driven models, the logic that we can drive at the highest may be of abstraction. It might not be same when we try to modularize it. It is so because the logic is guided by the weight and activation functions. Moreover, DL systems are behaviour-driven systems which are built by executing training program on training data. Here, underlying logic is guided by the training data and not the requirement (as in traditional software).

IX. Hypothesis

Let us make some assumption about the samples, adversarial samples and normal samples. In testing, adversarial samples are those samples which are vulnerable to any changes and show far more deviation in behaviour with respect to usual samples. Consider a scenario where the original DNN models have undergone a set of transformation rules to yield mutated DNN models. These mutated DNNs usually tend to label an adversarial data with a different label (label generated by original mutated DNN). We would assume this state, and try to measure the crucial factors such as model uncertainty estimate, density estimate, model sensitivity to the input changes.

Even before we can create a procedure for our hypothesis, we need an efficient way to generate the mutants. The fundamental approach is to generate or seed several program level mutations (mutants). This would require program under consideration to go through set of mutation operators by applying set of transformation rules. To the core of which lies, the process to define mutant operators. As it is known that traditional software systems are logic oriented, structured, while DNN models are behaviour and model oriented, therefore mutation operators' application to the former scenario (Traditional Software Systems) does not work for the latter (DNN Systems). There are quite a few techniques which work independently and using mutation testing in order to establish the testing framework.

X. Approach

Initialization

One of the initial approaches would be a way to build foundation steps to measure Label Change Rate (LCR) for the adversarial samples and normal samples. This is measurable when we inject these samples into a set of already mutated DNN models.

TABLE I. Model Mutation Operator

Mutation Operator	**Level**	**Description**
Gaussian Fuzzing (GF)	Weight	Fuzzy weight by Gaussian Distribution
Weight Shuffling (WS)	Neuron	Shuffle selected weights
Neuron Switch (NS)	Neuron	Switch two neurons within a layer
Neuron Activation Inverse (NAI)	Neuron	Change the activation status of a neuron

Let *x: input sample* (adversarial sample or normal sample).
Let *f: DNN model* (post mutation operators are applied).

Now, we go through the model mutation operator as provided in the Table 1 (sequence wise) and select the mutation models. Quite a few times, the output mutated model is of moderate to low quality (assuming high precision and confidence as measure of high quality mutated models). This means that the accuracy and effectiveness on the training data work well, however on the test data, it significantly deprecates. We let go or ignore these low quality mutated models. Only mutated models with high accuracy are considered. We can adopt the scale based on our experience and historical data obtained from mutated models. Ideally, any model with more that 90% accuracy of the original model is part of the set. This is to make sure that we meet the decision boundary [25] conditions and they are not impacted much. Upon segregating the mutated models, we further obtain a label of the input sample on each mutated model.

Building a Model

In this stage, we follow the hypothesis to create a model. This model validates (on certain criteria) the observation. If we recount, earlier we mentioned that adversarial samples are generated in such a way that it tends to minimize the mutated behaviour on normal samples, while, it is being able to jump the decision boundary [25]. There are different ways of mutation to achieve this behaviour. As per the hypothesis, the effective adversarial samples are closer to the decision boundary. This minimizes the restricted modification in the model. With this, adversarial samples would be considered as a case of crossing the decision boundary, unlike randomly selected mutated model. This implies, if we inject mutated adversarial sample into the mutated model, the outcome of the label tends to change it from its original label.

Algorithm Design

Experiments and test results show that LCR can be a distinguisher between adversarial samples and normal samples. We can discuss on the algorithm which can be designed to detect samples at runtime based on LCR measures of the provided samples. This algorithm would delete the LCR, and would keep on generating more effective and accurate mutated models. For this to happen, we must define a stopping condition on the mutation model generation algorithm that could be satisfied. Prediction algorithm can help us get a set of mutated models with higher accuracy beforehand.

XI. Conclusion and Future Work

In this work, we proposed the machine learning mutation testing framework, its usefulness and approach to detect adversarial samples for DNN at runtime. We laid down the details of source level mutation techniques on datasets (training and test) and training (or test) programs. This required us to further the details of the process and techniques involved in designing source level mutation operators, and feed faults into the DNN models during their development and testing process. This accompanied the details of model level mutation technique. Model level mutation technique differs from the source level mutation technique in the approach that it adopts to inject the faults. Model level mutation technique directly feeds the faults into the DNN system. It is also noteworthy how to measure the quality of these mutation models.

We also briefly touched upon how to predict the mutant operators even before we can analyse the same by executing. This is primarily done as mutants' generation is a computationally expensive approach. In the end, we proposed a hypothesis and an approach to build the problem set, analyse it and proceed under certain assumption to mitigate the same.

References

[1] G. Litjens et al., "A survey on deep learning in medical image analysis," Medical Image Analysis, vol. 42, pp. 60–88, 2017.

[2] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems," in 26th Symposium on Operating Systems Principles, 2017, pp. 1–18.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving," in 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 2722–2730.

[4] P. Ammann and J. Offutt, Introduction to Software Testing, 2nd ed., New York: Cambridge University Press, 2016.

[5] M. Pacula, "Unit-Testing Statistical Software," Maciej Pacula. Updated February 17, 2011. [Blog]. Available: http://blog. mpacula.com/2011/02/17/unit-testing-statistical-software.

[6] A. Ramanathan, L. L. Pullum, F. Hussain, D. Chakrabarty, and S. K. Jha, "Integrating symbolic and statistical methods for testing intelligent systems: Applications to machine learning and computer vision," in 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016, pp. 786-791.

[7] S. Amershi, A. Begel, C. Bird, R. DeLine, H. G., E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software Engineering for Machine Learning: A Case Study," 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291-300.

[8] M. Mohri, A. Rostamizadeh, and A. Talwalkar, Foundations of Machine Learning, 2nd ed., Cambridge: The MIT Press, 2018.

[9] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," Science, vol. 349, no. 6245, pp. 255–260, 2015.

[10] T. M. Mitchell, Machine Learning, New York: McGraw-Hill, 1997.

[11] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507–525, 2015.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[13] J. Shawe-Taylor and N. Cristianini, "Kernel Methods for Pattern Analysis," Cambridge: Cambridge University Press, 2004.

[14] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," IEEE Transactions on Systems, Man, and Cybernetics, vol. 21, no. 3, pp. 660–674, 1991.

[15] M. Liu, T. Breuel, and J. Kautz, "Unsupervised Image-to-Image Translation Networks," in 31st International Conference on Neural Information Processing Systems (NIPS 2017), 2017, pp. 700–708.

[16] X. Pan, Y. You, Z. Wang, and C. Lu, "Virtual to Real Reinforcement Learning for Autonomous Driving," arXiv:1704.03952 [cs.AI], 2017.

[17] D. Clark and R. M. Hierons, "Squeeziness: An information theoretic measure for avoiding fault masking," Information Processing Letters, vol. 112, no. 8–9, pp. 335–340, 2012.

[18] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," in 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008, pp. 249-258.

[19] "IEEE Standard Glossary of Software Engineering Terminology," in IEEE Std 610.12-1990, pp.1-84, 1990.

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

[21] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, "Applied Linear Statistical Models," 4th ed., Chicago: Irwin, 1996.

[22] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-Guided Black-Box Safety Testing of Deep Neural Networks," in 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018), 2018, pp. 408–426.

[23] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software Engineering for Machine Learning: A Case Study," in 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291–300.

[24] 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 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), 2018, pp. 100-111.

[25] W. Shen, Y. Li, Y. Han, L. Chen, D. Wu, Y. Zhou, and B. Xu, "Boundary sampling to boost mutation testing for deep learning models," Information and Software Technology, vol. 130, Article ID 106413, 2021.