EDITOR **PHIL LAPLANTE**
Penn State; plaplante@psu.edu

On the Testability of Artificial Intelligence and Machine Learning Systems

Raghvinder S. Sangwan, Youakim Badr, Satish Srinivasan, and Partha Mukherjee, Pennsylvania State University

This article examines current testing techniques for the quality assurance of artificial intelligence and machine learning systems. It organizes them based on the granularity of testing level and explores design tactics using these techniques.

Artificial intelligence (AI) and machine learning (ML) systems (referred to as AI systems in the rest of the article) continuously adapt their behavior by learning from the environment in which they operate. The dependability of these systems, in part, relies on quality assurance. However, traditional approaches to software testing are limited because the logic in AI systems, unlike traditional systems, is determined by the data used in training them and the stochastic nature of the learning process that makes them nondeterministic. Repetition of training yields different outcomes with a likelihood of an unintended behavior that can lead to a fault or failure.[1] Testing approaches for explicitly checking behaviors that we expect AI systems to follow are challenged by the following:[1,2]

- *Lack of test oracles*: AI systems reason probabilistically; hence, their outputs are learned and predicted by an ML model rather than one specified prior to testing.
- *Large input space*: It is difficult to determine a test data set that is representative of a voluminous and diverse input space.

Digital Object Identifier 10.1109/MC.2021.3132710
Date of current version: 11 March 2022

- *High white-box testing effort*: Testing all possible states within ML models of an AI system is intractable.

In the absence of oracles combined with a large input and state space, tests can only detect crashes (smoke testing).

Software testing, however, is an important task within the software development lifecycle and aims at providing stakeholders with measurable indicators about the quality of the software product under design. Traditional software testing techniques and tactics could be applied to the development of AI systems in which programming languages are intrinsic components. Nevertheless, testing AI systems is not just limited to software bugs and faults. They raise specific problems related to their quality and must be tested to determine their correctness under the assumption of some specific hypotheses (statistical and probabilistic) and verified to ensure that their outcomes fit the expected behavior.

TECHNIQUES FOR TESTING AI SYSTEMS

Table 1 organizes several techniques that have been proposed for testing AI systems[1,3] into four distinct categories according to the level of granularity of the entity being tested within the system. Input testing analyzes the training data for potential reasons that can lead to unsuccessful training, such as corner cases and for out-of-distribution or underrepresented data in the initial training set.

> Repetition of training yields different outcomes with a likelihood of an unintended behavior that can lead to a fault or failure.

Model testing uses measures of accuracy for classifiers or mean-square errors for regressors to look for a suboptimal model architecture, its training process, and hyperparameters. Model testing can be used to identify inputs for which the model produces wrong predictions. Perturbing inputs should either have no effect (invariance) or a predictable effect (directional expectation) on the model performance. Input perturbations can also generate adversarial examples to test the robustness of ML models. Data slicing can be used for quantifying model performance for specific subsets of the input data. Traditional techniques such as equivalence class partitioning and boundary value analysis can also be used to select inputs for testing a model. Similar to code coverage in traditional software systems, model coverage has been applied to systematically exercise different parts of a model. Concolic testing has been used for more efficient and effective model coverage. Researchers have also experimented with mutation testing with some success. Metamorphic testing has been used to overcome the issue of lack of test oracles.

Although individual models and software components may behave correctly when tested in isolation, integration testing exercises them together to uncover issues that emerge from their interactions during their deployment in real-world scenarios. Search-based testing has been used in autonomous vehicles to detect undesirable feature interactions.

System testing is important to validate the behavior of a complete system in its operational environment. Search-based testing has been used for testing advanced driver assistance systems and vision-based control systems. Adversarial testing has been used for the evaluation of self-driving software in autonomous vehicles and driver assistance systems. Metamorphic testing has been used for detecting failures when multiple ML-based components interact with each other in activity recognition chain systems.

TABLE 1. Testing at different levels of granularity.

Input testing	Model testing	Integration testing	System testing
• Corner-case analysis • Out-of-distribution or underrepresented input	• Input perturbation • Adversarial testing • Data slicing • Equivalence class partitioning • Boundary value analysis • Model coverage • Concolic testing • Mutation testing • Metamorphic testing	• Search-based testing	• Search-based testing • Adversarial testing • Metamorphic testing

DESIGN TACTICS FOR IMPROVING THE TESTABILITY OF AI SYSTEMS

Table 2 summarizes concerns from a software and systems engineering perspective that need to be addressed when designing AI systems. These issues have been derived from five research problems discussed by Amodei et al.[4] that highlight risks associated with poorly designed AI systems that, if manifested, could lead

to undesirable outcomes, including accidents that result in loss of life or property.

These concerns have been illustrated[4] using the example of a cleaning robot. A robot should not knock off items that it should not disturb while cleaning the room (avoiding a negative side effect); it should not compromise its vision system so it does not have to clean when it can't see the mess (avoiding reward hacking); it should not throw away valuable items, such as a cell phone on the floor, because it has limited information on what constitutes trash (scalable supervision concern); while deciding to mop a room it should not put a mop in an electrical outlet (safe exploration concern); and it should be able to adapt its cleaning strategies learned in an office to cleaning a factory floor (showing robustness to distributed shift). Testing strategies need to be in place to ensure that an AI system is not vulnerable to these risks, and that it behaves as intended.

Using this software and systems engineering perspective, we are methodically exploring and addressing the design concerns listed in Table 2 using design tactics for testability.[5] As shown in Figure 1, the goal of these tactics is to allow ease of testing by controlling and observing the behavior of an AI system.

The input into an AI system, its output, and its internal state are monitored through a test monitor that uses them to validate the system's behavior. At the time of this writing, we have begun to examine the scalable supervision concern associated with the objective function evaluation category as well as the safe exploration and robustness to distributional shift concerns associated with the behavior during learning process category (see Table 2). As an illustrative example, we will use a predictive analytics system for adjudicating loans to applicants while minimizing the risk of losing money to demonstrate the use of design tactics for testability to address these concerns.

Figure 2 shows the different elements of the predictive analytics system used in our example. A web-based loan app takes a loan request from a bank customer. It sends the request to a predictive analysis ML model through the StreamLit framework. The ML model makes a prediction and sends its response back to the user through the StreamLit framework. The framework also forwards the original user request to a test monitor. The monitor uses its monitor behavior component to validate the runtime behavior of the ML model and also writes the user request (input data) along with the model prediction (output data) to a logger, which saves these data to a persistent store. The monitor also has a monitor drift component, which is scheduled to run periodically on the aggregated user input data for that period to see if the model performance has drifted to a level where it may need retraining. Additionally, the monitor's run self-diagnostics component performs diagnostics on the ML model on a periodic schedule to ensure that the retrained model continues to operate as expected.

> Similar to code coverage in traditional software systems, model coverage has been applied to systematically exercise different parts of a model.

The ML model is continuously validated at runtime by the monitor

TABLE 2. Design concerns for AI systems.

Category	Concern	Description
Objective function	Avoid negative side effects	An objective function must not focus only on accomplishing some specific task in the environment but should also take into account other variables in the (potentially very large) environment, ignoring of which might actually be harmful
	Avoid reward hacking	Prevent gaming of the objective function
Objective function evaluation	Scalable supervision	Avoid harmful behavior due to bad extrapolations from limited samples intended to avoid the cost of evaluating an expensive objective function
Behavior during learning process	Safe exploration	Learn new behavior with no undesirable consequences
	Robustness to distributional shift	Continue to operate as intended in an environment

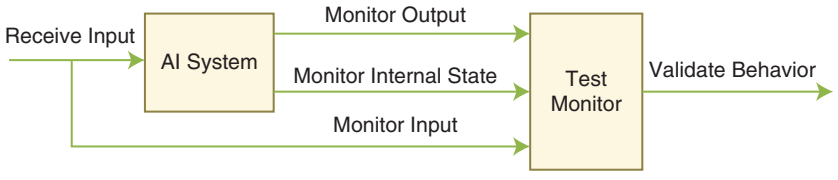

FIGURE 1. The use of a test monitor for improving testability in an AI system.

behavior component using metamorphic testing, a software testing technique that uses metamorphic relations to address the test oracle problem discussed earlier. It verifies and validates the "expected" relationships between inputs and outputs across multiple executions of the loan app. For instance, given a certain income level, years of employment, and debt-to-income ratio as input, there is a high probability that the loan app will approve the loan. If this relationship is violated across multiple executions of the system, then that indicates a fault.

We also use the test monitor to observe the effect when input data are changed in known ways (a perturb input tactic). For instance, whenever a new loan application is submitted by the user, the monitor behavior component may invoke the ML model after transformation on one or more predictor variables (such as gender, age, or income). The differences in the prediction on the original input and the transformed input are observed to see if there is no effect on the model prediction (the ML model is invariant to the change) or a predictable effect (the ML model responds to the change in known ways). For example, simply changing the gender of a loan applicant should have no effect, but changing the income level should have a predictable effect on the approval of the loan.

Periodically, the monitor drift component analyzes the input data from the user logged by the test monitor to check whether the model performance has degraded over time because of any changes in the environment that violate the model assumptions (a measure model accuracy tactic). For instance, the model performance is believed to degrade when the distribution of the input data set is different from the distribution of the training data set. If the model is determined to be drifting over the period, then model retraining needs to be performed. To ensure that the training and input data sets have the same distribution, we employ the Kolmogorov–Smirnov (K–S) testing tactic.

In addition to monitoring the distribution of the predictors and the response variable, the monitor drift component also performs pairwise correlation between the features in these data sets. Here, we use the adjusted *r*-squared values as indicators for the pairwise correlation between the features. To determine whether the test and the training data sets are from the same or different distributions, we compute the cosine similarity index on the obtained adjusted *r*-squared values.

The run self-diagnostics component periodically performs testing using equivalence class partitioning, boundary value analysis, and data slicing. For instance, the aggregated user input data can be partitioned into different equivalence classes. Each class basically consists of instances that have a similar categorical type value for a given predictor. For example, a partition class would include all input instances where the gender is "male," the employment duration is "long," and the income level is "high."

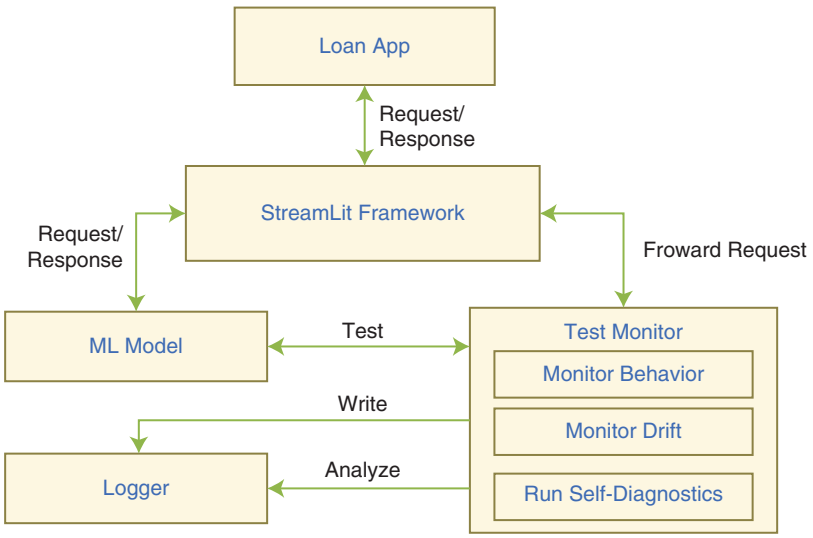

FIGURE 2. A predictive analytics system used for adjudicating loan applications.

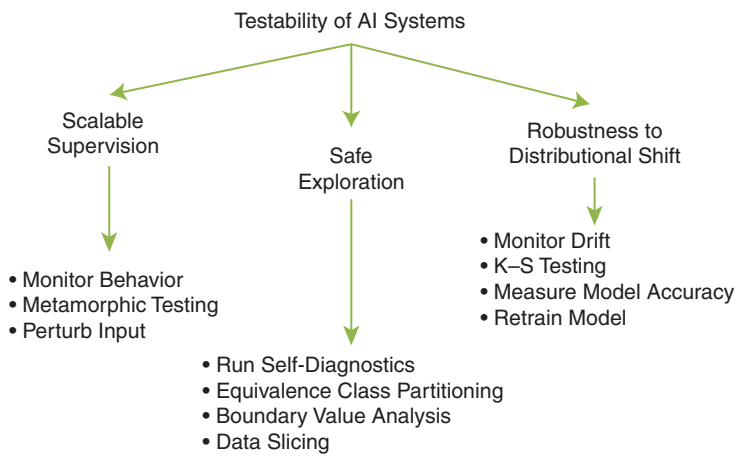

FIGURE 3. A testability design tactics catalog for AI systems.

Knowing the test oracle for each class, the run self-diagnostics component can execute the ML model on any random sample from a given equivalence class or on samples at the boundary of a class. These types of tests can be performed on any input data slice or subset as well.

We conducted several experiments during runtime to test the performance of the ML model. Whenever a new user input (new test data) comes into the system, the monitor behavior component performs a metamorphic testing or testing on perturbed input, as discussed earlier. In all of the experiments, the monitor drift component was instructed to keep monitoring the p value of the K–S test and the cosine similarity index. Any indication that the p value is less than 0.05 and the cosine similarity index is closer to 0 prompts the test monitor to invoke the model retraining step. Retraining is also necessary when the model accuracy drops. The run self-diagnostics component ensures that the model (original or retrained) will continue to behave as expected by self-diagnosing itself against known classes or slices of data.

This motivational example demonstrates how testability design tactics can be used for controlling and observing the behavior of an AI system at runtime to monitor the effectiveness of the system once it has been deployed. We have begun organizing these tactics into a testability design tactics catalog, as shown in Figure 3.

The run self-diagnostics component ensures that the model (original or retrained) will continue to behave as expected by self-diagnosing itself against known classes or slices of data.

The root of this hierarchy is the testability of AI systems, and the intermediate nodes represent categories of concerns related to this testability that the tactics at the leaf level address. Not all testability design concerns (such as those enumerated in Table 2) have been addressed in the catalog at this time. It is the intent of the authors to continue to broaden this catalog with additional design tactics as they are discovered. C

ACKNOWLEDGMENT

This material is based upon work funded and supported by the 2020 IndustryXchange Multidisciplinary Research Seed Grant from Pennsylvania State University.

REFERENCES

1. V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: A systematic mapping," *Empirical Softw. Eng.*, vol. 25, no. 6, pp. 5193–5254, 2020, doi: 10.1007/s10664-020-09881-0.
2. D. Marijan, A. Gotlieb, and M. Kumar Ahuja, "Challenges of testing machine learning based systems," in *Proc. 2019 IEEE Int. Conf. Artif. Intell. Testing (AITest)*, Newark, CA, USA, pp. 101–102, doi: 10.1109/AITest.2019.00010.
3. J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Trans. Softw. Eng.*, early access, Feb. 2020, doi: 10.1109/TSE.2019.2962027.
4. D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mane, "Concrete problems in AI safety," 2016, *arXiv:1606.06565*.
5. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2012.

RAGHVINDER S. SANGWAN is an associate professor of software engineering in the School of Graduate Professional Studies, Pennsylvania State University, Malvern, Pennsylvania, 19355, USA. Contact him at rsangwan@psu.edu.

YOUAKIM BADR is an associate professor of data analytics in the School of Graduate Professional Studies, Pennsylvania State University, Malvern, Pennsylvania, 19355, USA. Contact him at yzb61@psu.edu.

SATISH SRINIVASAN is an associate professor of information science in the School of Graduate Professional Studies, Pennsylvania State University, Malvern, Pennsylvania, 19355, USA. Contact him at sus64@psu.edu.

PARTHA MUKHERJEE is an assistant professor of data analytics in the School of Graduate Professional Studies, Pennsylvania State University, Malvern, Pennsylvania, 19355, USA. Contact him at pom5109@psu.edu.