Machine Learning Testing: Survey, Landscapes and Horizons

Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu

Abstract—This paper provides a comprehensive survey of techniques for testing machine learning systems; Machine Learning Testing (ML testing) research. It covers 144 papers on testing properties (e.g., correctness, robustness, and fairness), testing components (e.g., the data, learning program, and framework), testing workflow (e.g., test generation and test evaluation), and application scenarios (e.g., autonomous driving, machine translation). The paper also analyses trends concerning datasets, research trends, and research focus, concluding with research challenges and promising research directions in ML testing.

Index Terms—Machine learning, software testing, deep neural network

1 Introduction

The prevalent applications of machine learning arouse natural concerns about trustworthiness. Safety-critical applications such as self-driving systems [1], [2] and medical treatments [3], increase the importance of behaviour relating to correctness, robustness, privacy, efficiency and fairness. Software testing refers to any activity that aims to detect the differences between existing and required behaviour [4]. With the recent rapid rise in interest and activity, testing has been demonstrated to be an effective way to expose problems and potentially facilitate to improve the trustworthiness of machine learning systems.

For example, DeepXplore [1], a differential white-box testing technique for deep learning, revealed thousands of incorrect corner case behaviours in autonomous driving learning systems; Themis [5], a fairness testing technique for detecting causal discrimination, detected significant ML model discrimination towards gender, marital status, or race for as many as 77.2 percent of the individuals in datasets to which it was applied.

In fact, some aspects of the testing problem for machine learning systems are shared with well-known solutions already widely studied in the software engineering literature. Nevertheless, the statistical nature of machine learning systems and their ability to make autonomous decisions raise additional, and challenging, research questions for software testing [6], [7].

- *Jie M. Zhang is with CREST, University College London, London WC1E 6BT, U.K. E-mail: jie.zhang@ucl.ac.uk.*
- *Mark Harman is with CREST, University College London, London WC1E 6BT, U. K., and also with Facebook, London W1T 1FB, U. K. E-mail: mark.harman@ucl.ac.uk.*
- *Lei Ma is with Kyushu University, Fukuoka 819-0395, Japan. E-mail: malei@ait.kyushu-u.ac.jp.*
- *Yang Liu is with Nanyang Technological University, Singapore 639798. E-mail: yangliu@ntu.edu.sg.*

Manuscript received 23 June 2019; revised 6 Nov. 2019; accepted 24 Nov. 2019. Date of publication 17 Feb. 2020; date of current version 10 Jan. 2022. (Corresponding author: Jie M. Zhang.) Recommended for acceptance by Y. Brun. Digital Object Identifier no. 10.1109/TSE.2019.2962027

Machine learning testing poses challenges that arise from the fundamentally different nature and construction of machine learning systems, compared to traditional (relatively more deterministic and less statistically-orientated) software systems. For instance, a machine learning system inherently follows a data-driven programming paradigm, where the decision logic is obtained via a training procedure from training data under the machine learning algorithm's architecture [8]. The model's behaviour may evolve over time, in response to the frequent provision of new data [8]. While this is also true of traditional software systems, the core underlying behaviour of a traditional system does not typically change in response to new data, in the way that a machine learning system can.

Testing machine learning also suffers from a particularly pernicious instance of the *Oracle Problem* [9]. Machine learning systems are difficult to test because they are designed to provide an answer to a question for which no previous answer exists [10]. As Davis and Weyuker said [11], for these kinds of systems 'There would be no need to write such programs, if the correct answer were known'. Much of the literature on testing machine learning systems seeks to find techniques that can tackle the Oracle problem, often drawing on traditional software testing approaches.

The behaviours of interest for machine learning systems are also typified by emergent properties, the effects of which can only be fully understood by considering the machine learning system as a whole. This makes testing harder, because it is less obvious how to break the system into smaller components that can be tested, as units, in isolation. From a testing point of view, this emergent behaviour has a tendency to migrate testing challenges from the unit level to the integration and system level. For example, low accuracy/precision of a machine learning model is typically a composite effect, arising from a combination of the behaviours of different components such as the training data, the learning program, and even the learning framework/library [8].

Errors may propagate to become amplified or suppressed, inhibiting the tester's ability to decide where the fault lies.

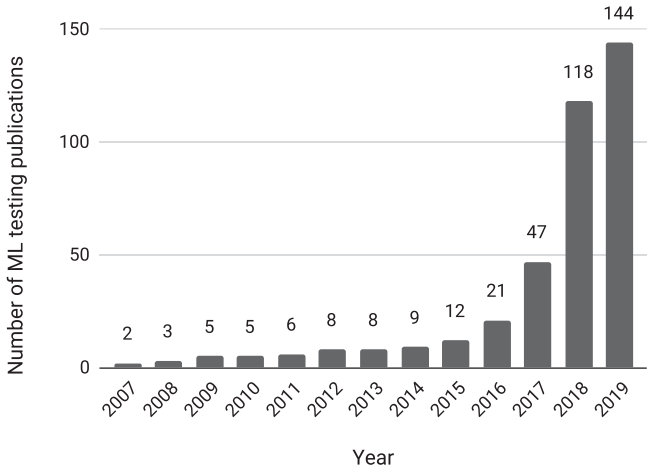


Fig. 1. Machine learning testing publications (accumulative) during 2007-2019.

These challenges also apply in more traditional software systems, where, for example, previous work has considered failed error propagation [12], [13] and the subtleties introduced by fault masking [14], [15]. However, these problems are far-reaching in machine learning systems, since they arise out of the nature of the machine learning approach and fundamentally affect all behaviours, rather than arising as a side effect of traditional data and control flow [8].

For these reasons, machine learning systems are thus sometimes regarded as '*non-testable*' software. Rising to these challenges, the literature has seen considerable progress and a notable upturn in interest and activity: Fig. 1 shows the cumulative number of publications on the topic of testing machine learning systems between 2007 and June 2019 (we introduce how we collected these papers in Section 4.2). From this figure, we can see that 85 percent of papers have appeared since 2016, testifying to the emergence of new software testing domain of interest: machine learning testing.

In this paper, we use the term '*Machine Learning Testing*' (*ML testing*) to refer to any activity aimed at detecting differences between existing and required behaviours of machine learning systems. ML testing is different from testing approaches that use machine learning or those that are guided by machine learning, which should be referred to as 'machine learning-based testing'. This nomenclature accords with previous usages in the software engineering literature. For example, the literature uses the terms 'state-based testing' [16] and 'search-based testing' [17], [18] to refer to testing techniques that make use of concepts of state and search space, whereas we use the terms 'GUI testing' [19] and 'unit testing' [20] to refer to test techniques that tackle challenges of testing Graphical User Interfaces (GUIs) and code units.

This paper seeks to provide a comprehensive survey of ML testing. We draw together the aspects of previous work that specifically concern software testing, while simultaneously covering all types of approaches to machine learning that have hitherto been tackled using testing. The literature is organised according to four different aspects: the testing properties (such as correctness, robustness, and fairness), machine learning components (such as the data, learning program, and framework), testing workflow (e.g., test generation, test execution, and test evaluation), and application scenarios (e.g., autonomous driving and machine translation). Some papers address multiple aspects. For such papers, we mention them in all the aspects correlated (in different sections). This ensures that each aspect is complete.

Additionally, we summarise research distribution (e.g., among testing different machine learning categories), trends, and datasets. We also identify open problems and challenges for the emerging research community working at the intersection between techniques for software testing and problems in machine learning testing. To ensure that our survey is self-contained, we aimed to include sufficient material to fully orientate software engineering researchers who are interested in testing and curious about testing techniques for machine learning applications. We also seek to provide machine learning researchers with a complete survey of software testing solutions for improving the trustworthiness of machine learning systems.

There has been previous work that discussed or surveyed aspects of the literature related to ML testing. Hains *et al.* [21], Ma *et al.* [22], and Huang *et al.* [23] surveyed secure deep learning, in which the focus was deep learning security with testing being one of the assurance techniques. Masuda *et al.* [24] outlined their collected papers on software quality for machine learning applications in a short paper. Ishikawa [25] discussed the foundational concepts that might be used in any and all ML testing approaches. Braiek and Khomh [26] discussed defect detection in machine learning data and/or models in their review of 39 papers. As far as we know, no previous work has provided a comprehensive survey particularly focused on machine learning testing.

In summary, the paper makes the following contributions:

1) *Definition*. The paper defines Machine Learning Testing (ML testing), overviewing the concepts, testing workflow, testing properties, and testing components related to machine learning testing.
2) *Survey*. The paper provides a comprehensive survey of 144 machine learning testing papers, across various publishing areas such as software engineering, artificial intelligence, systems and networking, and data mining.
3) *Analyses*. The paper analyses and reports data on the research distribution, datasets, and trends that characterise the machine learning testing literature. We observed a pronounced imbalance in the distribution of research efforts: among the 144 papers we collected, around 120 of them tackle supervised learning testing, three of them tackle unsupervised learning testing, and only one paper tests reinforcement learning. Additionally, most of them (93) centre on correctness and robustness, but only a few papers test interpretability, privacy, or efficiency.
4) *Horizons*. The paper identifies challenges, open problems, and promising research directions for ML testing, with the aim of facilitating and stimulating further research.

Fig. 2 depicts the paper structure. More details of the review schema can be found in Section 4.

2 Preliminaries of Machine Learning

This section reviews the fundamental terminology in machine learning so as to make the survey self-contained.

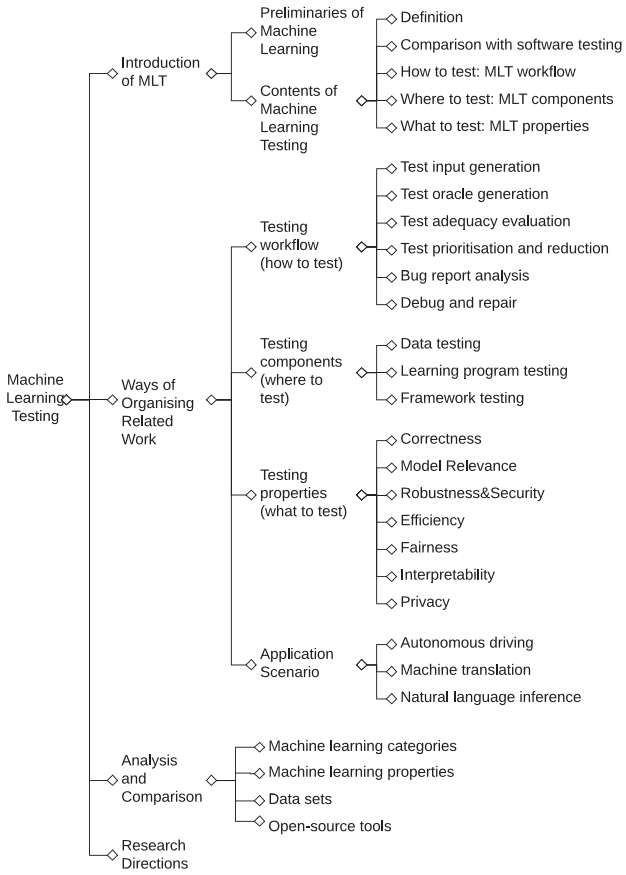


Fig. 2. Tree structure of the contents in this paper.

Machine Learning (ML) is a type of artificial intelligence technique that makes decisions or predictions from data [27], [28]. A machine learning system is typically composed from following elements or terms.

Dataset: A set of instances for building or evaluating a machine learning model.

At the top level, the data could be categorised as:

- *Training data*: the data used to 'teach' (train) the algorithm to perform its task.
- *Validation data*: the data used to tune the hyper-parameters of a learning algorithm.
- *Test data*: the data used to validate machine learning model behaviour.

Learning program: the code written by developers to build and validate the machine learning system.

Framework: the library, or platform being used when building a machine learning model, such as *Pytorch* [29], *TensorFlow* [30], *Scikit-learn* [31], *Keras* [32], and *Caffe* [33].

In the remainder of this section, we give definitions for other ML terminology used throughout the paper.

Instance: a piece of data recording the information about an object.

Feature: a measurable property or characteristic of a phenomenon being observed to describe the instances.

Label: value or category assigned to each data instance.

Test error: the difference ratio between the real conditions and the predicted conditions.

Generalisation error: the expected difference ratio between the real conditions and the predicted conditions of any valid data.

Model: the learned machine learning artefact that encodes decision or prediction logic which is trained from the training data, the learning program, and frameworks.

There are different types of machine learning. From the perspective of training data characteristics, machine learning includes:

Supervised learning: a type of machine learning that learns from training data with labels as learning targets. It is the most widely used type of machine learning [34].

Unsupervised learning: a learning methodology that learns from training data without labels and relies on understanding the data itself.

Reinforcement learning: a type of machine learning where the data are in the form of sequences of actions, observations, and rewards, and the learner learns how to take actions to interact in a specific environment so as to maximise the specified rewards.

Let $\mathcal{X} = (x_1, \ldots, x_m)$ be the set of unlabelled training data. Let $\mathcal{Y} = (c(x_1), \ldots, c(x_m))$ be the set of labels corresponding to each piece of training data x_i. Let concept $C : \mathcal{X} \to \mathcal{Y}$ be the mapping from $\mathcal{X}$ to $\mathcal{Y}$ (the real pattern). The task of supervised learning is to learn a mapping pattern, i.e., a model, h based on $\mathcal{X}$ and $\mathcal{Y}$ so that the learned model h is similar to its true concept C with a very small generalisation error. The task of unsupervised learning is to learn patterns or clusters from the data without knowing the existence of labels $\mathcal{Y}$.

Reinforcement learning guides and plans with the learner actively interacting with the environment to achieve a certain goal. It is usually modelled as a Markov decision process. Let S be a set of states, A be a series of actions. Let s and s' be two states. Let $r_a(s, s')$ be the reward after transition from s to s' with action $a \in A$. Reinforcement learning is to learn how to take actions in each step to maximise the target awards.

Machine learning can be applied to the following typical tasks [28]:[1]

1) *Classification*: to assign a category to each data instance; E.g., image classification, handwriting recognition.
2) *Regression*: to predict a value for each data instance; E.g., temperature/age/income prediction.
3) *Clustering*: to partition instances into homogeneous regions; E.g., pattern recognition, market/image segmentation.
4) *Dimension reduction*: to reduce the training complexity; E.g., dataset representation, data pre-processing.
5) *Control*: to control actions to maximise rewards; E.g., game playing.

Fig. 3 shows the relationship between different categories of machine learning and the five machine learning tasks. Among the five tasks, classification and regression belong to supervised learning; Clustering and dimension reduction belong to unsupervised learning. Reinforcement learning is widely adopted to control actions, such as to control AI-game players to maximise the rewards for a game agent.

In addition, machine learning can be classified into *classic machine learning* and *deep learning*. Algorithms like Decision

1. These tasks are defined based on the nature of the problems solved instead of specific application scenarios such as language modelling.

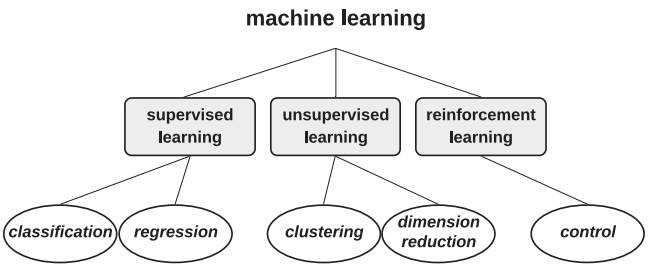


Fig. 3. Machine learning categories and tasks.

Tree [35], SVM [27], linear regression [36], and Naive Bayes [37] all belong to classic machine learning. Deep learning [38] applies Deep Neural Networks (DNNs) that uses multiple layers of nonlinear processing units for feature extraction and transformation. Typical deep learning algorithms often follow some widely used neural network structures like Convolutional Neural Networks (CNNs) [39] and Recurrent Neural Networks (RNNs) [40]. The scope of this paper involves both classic machine learning and deep learning.

3 Machine Learning Testing

This section gives a definition and analyses of ML testing. It describes the testing workflow (*how* to test), testing properties (*what* to test), and testing components (*where* to test).

3.1 Definition

A software bug refers to an imperfection in a computer program that causes a discordance between the existing and the required conditions [41]. In this paper, we refer the term 'bug' to the differences between existing and required behaviours of an ML system.[2]

Definition 1 (ML Bug). *An ML bug refers to any imperfection in a machine learning item that causes a discordance between the existing and the required conditions.*

We define ML testing as any activity aimed to detect ML bugs.

Definition 2 (ML Testing). *Machine Learning Testing refers to any activity designed to reveal machine learning bugs.*

The definitions of machine learning bugs and ML testing indicate three aspects of machine learning: the required conditions, the machine learning items, and the testing activities. A machine learning system may have different types of 'required conditions', such as correctness, robustness, and privacy. An ML bug may exist in the data, the learning program, or the framework. The testing activities may include test input generation, test oracle identification, test adequacy evaluation, and bug triage. In this survey, we refer to the above three aspects as testing properties, testing components, and testing workflow, respectively, according to which we collect and organise the related work.

Note that a test input in ML testing can be much more diverse in its form than that used in traditional software testing, because it is not only the code that may contain bugs, but also the data. When we try to detect bugs in data, one may even use a training program as a test input to check some properties required for the data.

2. The existing related papers may use other terms like 'defect' or 'issue'. This paper uses 'bug' as a representative of all such related terms considering that 'bug' has a more general meaning [41].

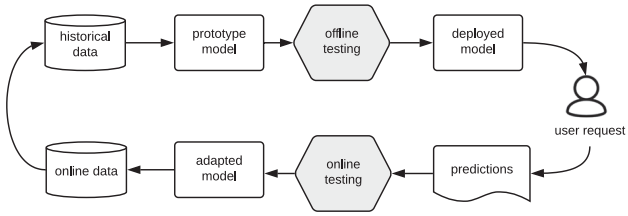


Fig. 4. Role of ML testing in ML system development.

3.2 ML Testing Workflow

ML testing workflow is about *how* to conduct ML testing with different testing activities. In this section, we first briefly introduce the role of ML testing when building ML models, then present the key procedures and activities in ML testing. We introduce more details of the current research related to each procedure in Section 5.

3.2.1 Role of Testing in ML Development

Fig. 4 shows the life cycle of deploying a machine learning system with ML testing activities involved. At the very beginning, a prototype model is generated based on historical data; before deploying the model online, one needs to conduct offline testing, such as cross-validation, to make sure that the model meets the required conditions. After deployment, the model makes predictions, yielding new data that can be analysed via online testing to evaluate how the model interacts with user behaviours.

There are several reasons that make online testing essential. First, offline testing usually relies on test data, while test data usually fails to fully represent future data [42]; Second, offline testing is not able to test some circumstances that may be problematic in real applied scenarios, such as data loss and call delays. In addition, offline testing has no access to some business metrics such as open rate, reading time, and click-through rate.

In the following, we present an ML testing workflow adapted from classic software testing workflows. Fig. 5 shows the workflow, including both offline testing and online testing.

3.2.2 Offline Testing

The workflow of offline testing is shown by the top dotted rectangle of Fig. 5. At the very beginning, developers need to conduct requirement analysis to define the expectations of the users for the machine learning system under test. In requirement analysis, specifications of a machine learning system are analysed and the whole testing procedure is planned. After that, test inputs are either sampled from the collected data or generated based on a specific purpose. Test oracles are then identified or generated (see Section 5.2 for more details of test oracles in machine learning). When the tests are ready, they need to be executed for developers to collect results. The test execution process involves building a model with the tests (when the tests are training data)

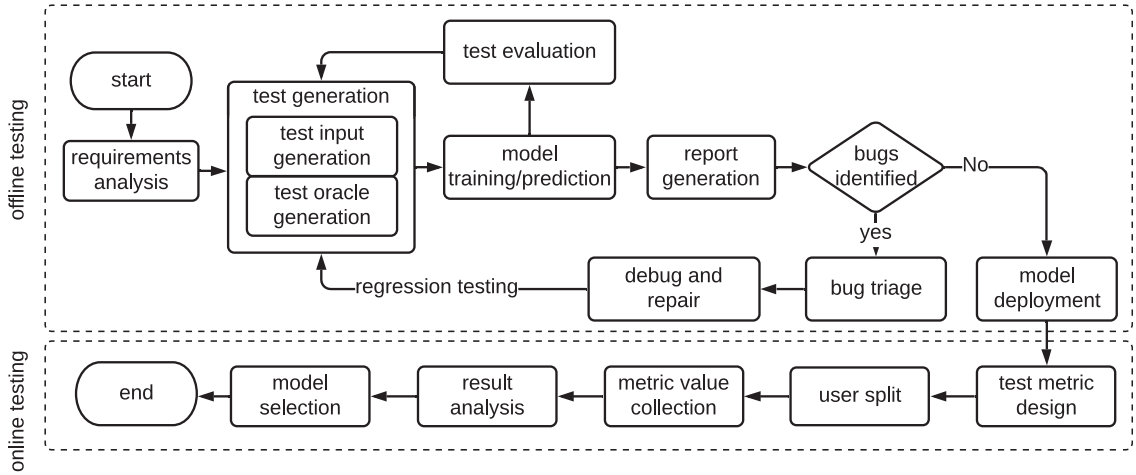


Fig. 5. Idealised Workflow of ML testing.

or running a built model against the tests (when the tests are test data), as well as checking whether the test oracles are violated. After the process of test execution, developers may use evaluation metrics to check the quality of tests, i.e., the ability of the tests to expose ML problems.

The test execution results yield a bug report to help developers to duplicate, locate, and solve the bug. Those identified bugs will be labelled with different severity and assigned for different developers. Once the bug is debugged and repaired, regression testing is conducted to make sure the repair solves the reported problem and does not bring new problems. If no bugs are identified, the offline testing process ends, and the model is deployed.

3.2.3 Online Testing

Offline testing tests the model with historical data without in the real application environment. It also lacks the data collection process of user behaviours. Online testing complements the shortage of offline testing, and aims to detect bugs after the model is deployed online.

The workflow of online testing is shown by the bottom of Fig. 5. There are different methods of conducting online testing for different purposes. For example, runtime monitoring keeps checking whether the running ML systems meet the requirements or violate some desired runtime properties. Another commonly used scenario is to monitor user responses, based on which to find out whether the new model is superior to the old model under certain application contexts. A/B testing is one typical type of such online testing [43]. It splits customers to compare two versions of the system (e.g., web pages). When performing A/B testing on ML systems, the sampled users will be split into two groups using the new and old ML models separately.

MAB (Multi-Armed Bandit) is another online testing approach [44]. It first conducts A/B testing for a short time and finds out the best model, then put more resources on the chosen model.

3.3 ML Testing Components

To build a machine learning model, an ML software developer usually needs to collect data, label the data, design learning program architecture, and implement the proposed architecture based on specific frameworks. The procedure of machine learning model development requires interaction with several components such as data, learning program, and learning framework, while each component may contain bugs.

Fig. 6 shows the basic procedure of building an ML model and the major components involved in the process. Data are collected and pre-processed for use; the learning program is the code for running to train the model; the framework (e.g., Weka, scikit-learn, and TensorFlow) offers algorithms and other libraries for developers to choose from, when writing the learning program.

Thus, when conducting ML testing, developers may need to try to find bugs in every component including the data, the learning program, and the framework. In particular, error propagation is a more serious problem in ML development because the components are more closely bonded with each other than traditional software [8], which indicates the importance of testing each of the ML components. We introduce the bug detection in each ML component below:

Bug Detection in Data. The behaviours of a machine learning system largely depends on data [8]. Bugs in data affect the quality of the generated model, and can be amplified to yield more serious problems over a period a time [45]. Bug detection in data checks problems such as whether the data is sufficient for training or test a model (also called completeness of the data [46]), whether the data is representative of future data, whether the data contains a lot of noise such as biased labels, whether there is skew between training data and test data [45], and whether there is data poisoning [47] or adversary information that may affect the model's performance.

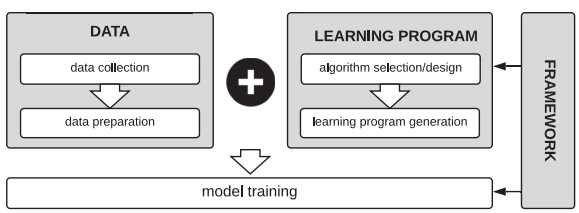


Fig. 6. Components (shown by the grey boxes) involved in ML model building.

Bug Detection in Frameworks. Machine Learning requires a lot of computations. As shown by Fig. 6, ML frameworks offer algorithms to help write the learning program, and platforms to help train the machine learning model, making it easier for developers to build solutions for designing, training and validating algorithms and models for complex problems. They play a more important role in ML development than in traditional software development. ML Framework testing thus checks whether the frameworks of machine learning have bugs that may lead to problems in the final system [48].

Bug Detection in Learning Program. A learning program can be classified into two parts: the algorithm designed by the developer or chosen from the framework, and the actual code that developers write to implement, deploy, or configure the algorithm. A bug in the learning program may arise either because the algorithm is designed, chosen, or configured improperly, or because the developers make typos or errors when implementing the designed algorithm.

3.4 ML Testing Properties

Testing properties refer to *what* to test in ML testing: for what conditions ML testing needs to guarantee for a trained model. This section lists some typical properties that the literature has considered. We classified them into basic functional requirements (i.e., correctness and model relevance) and non-functional requirements (i.e., efficiency, robustness,[3] fairness, interpretability).

These properties are not strictly independent of each other when considering the root causes, yet they are different external manifestations of the behaviours of an ML system and deserve being treated independently in ML testing.

3.4.1 Correctness

Correctness measures the probability that the ML system under test 'gets things right'.

Definition 3 (Correctness). *Let $\mathcal{D}$ be the distribution of future unknown data. Let x be a data item belonging to $\mathcal{D}$. Let h be the machine learning model that we are testing. $h(x)$ is the predicted label of x, $c(x)$ is the true label. The model correctness $E(h)$ is the probability that $h(x)$ and $c(x)$ are identical*

$$E(h) = Pr_{x\sim\mathcal{D}}[h(x) = c(x)]. \tag{1}$$

Achieving acceptable correctness is the fundamental requirement of an ML system. The real performance of an ML system should be evaluated on future data. Since future data are often not available, the current best practice usually splits the data into training data and test data (or training data, validation data, and test data), and uses test data to simulate future data. This data split approach is called cross-validation.

Definition 4 (Empirical Correctness). *Let $\mathcal{X} = (x_1, \ldots, x_m)$ be the set of unlabelled test data sampled from $\mathcal{D}$. Let h be the machine learning model under test. Let $\mathcal{Y}' = (h(x_1), \ldots, h(x_m))$ be the set of predicted labels corresponding to each training item x_i. Let $\mathcal{Y} = (y_1, \ldots, y_m)$ be the true labels, where each $y_i \in \mathcal{Y}$ corresponds to the label of $x_i \in \mathcal{X}$. The empirical correctness of model (denoted as $\hat{E}(h)$) is*

$$\hat{E}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{I}(h(x_i) = y_i), \tag{2}$$

where $\mathbb{I}$ is the indicator function; a predicate returns 1 if p is true, and returns 0 otherwise.

3.4.2 Model Relevance

A machine learning model comes from the combination of a machine learning algorithm and the training data. It is important to ensure that the adopted machine learning algorithm be not over-complex than just needed [51]. Otherwise, the model may fail to have good performance on future data, or have very large uncertainty.

The algorithm capacity represents the number of functions that a machine learning model can select (based on the training data at hand) as a possible solution. It is usually approximated by VC-dimension [52] or Rademacher Complexity [53] for classification tasks. VC-dimension is the cardinality of the largest set of points that the algorithm can shatter. Rademacher Complexity is the cardinality of the largest set of training data with fixed features that the algorithm shatters.

We define the relevance between a machine learning algorithm capacity and the data distribution as the problem of *model relevance* .

Definition 5 (Model Relevance). *Let $\mathcal{D}$ be the training data distribution. Let $R(\mathcal{D}, \mathcal{A})$ be the simplest required capacity of any machine learning algorithm $\mathcal{A}$ for $\mathcal{D}$. $R'(\mathcal{D}, \mathcal{A}')$ is the capacity of the machine learning algorithm $\mathcal{A}'$ under test. Model relevance is the difference between $R(\mathcal{D}, \mathcal{A})$ and $R'(\mathcal{D}, \mathcal{A}')$.*

$$f = |(R(\mathcal{D}, \mathcal{A}) - R'(\mathcal{D}, \mathcal{A}')|. \tag{3}$$

Model relevance aims to measure how well a machine learning algorithm fits the data. A low model relevance is usually caused by overfitting, where the model is too complex for the data, which thereby fails to generalise to future data or to predict observations robustly.

Of course, $R'(\mathcal{D}, \mathcal{A})$, the minimum complexity that is 'just sufficient' is hard to determine, and is typically approximate [42], [54]. We discuss more strategies that could help alleviate the problem of overfitting in Section 6.2.

3.4.3 Robustness

Robustness is defined by the IEEE standard glossary of software engineering terminology [55], [56] as: 'The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions'. Adopting a similar spirit to this definition, we define the robustness of ML as follows:

Definition 6 (Robustness). *Let S be a machine learning system. Let $E(S)$ be the correctness of S. Let $\delta(S)$ be the machine learning system with perturbations on any machine learning components such as the data, the learning program, or the framework. The robustness of a machine learning system is a measurement of the difference between $E(S)$ and $E(\delta(S))$*

$$r = E(S) - E(\delta(S)). \tag{4}$$

3. we adopt the more general understanding from software engineering community [49], [50], and regard robustness as a non-functional requirement.

Robustness thus measures the resilience of an ML system's correctness in the presence of perturbations.

A popular sub-category of robustness is called *adversarial robustness*. For adversarial robustness, the perturbations are designed to be hard to detect. Following the work of Katz *et al.* [57], we classify adversarial robustness into local adversarial robustness and global adversarial robustness. Local adversarial robustness is defined as follows.

Definition 7 (Local Adversarial Robustness). *Let x a test input for an ML model h. Let x' be another test input generated via conducting adversarial perturbation on x. Model h is δ-local robust at input x if for any x'.*

$$\forall x' : ||x - x'||_p \leq \delta \rightarrow h(x) = h(x'). \tag{5}$$

$||\cdot||_p$ represents p-norm for distance measurement. The commonly used p cases in machine learning testing are 0, 2, and ∞. For example, when $p = 2$, i.e. $||x - x'||_2$ represents the euclidean distance of x and x'. In the case of $p = 0$, it calculates the element-wise difference between x and x'. When $p = \infty$, it measures the the largest element-wise distance among all elements of x and x'.

Local adversarial robustness concerns the robustness at one specific test input, while global adversarial robustness measures robustness against all inputs. We define global adversarial robustness as follows.

Definition 8 (Global Adversarial Robustness). *Let x a test input for an ML model h. Let x' be another test input generated via conducting adversarial perturbation on x. Model h is ϵ-global robust if for any x and x'.*

$$\forall x, x' : ||x - x'||_p \leq \delta \rightarrow h(x) - h(x') \leq \epsilon. \tag{6}$$

3.4.4 Security

The security of an ML system is the system's resilience against potential harm, danger, or loss made via manipulating or illegally accessing ML components.

Security and robustness are closely related. An ML system with low robustness may be insecure: if it is less robust in resisting the perturbations in the data to predict, the system may more easily fall victim to adversarial attacks; For example, if it is less robust in resisting training data perturbations, it may also be vulnerable to data poisoning (i.e., changes to the predictive behaviour caused by adversarially modifying the training data).

Nevertheless, low robustness is just one cause of security vulnerabilities. Except for perturbations attacks, security issues also include other aspects such as model stealing or extraction. This survey focuses on the testing techniques on detecting ML security problems, which narrows the security scope to robustness-related security. We combine the introduction of robustness and security in Section 6.3.

3.4.5 Data Privacy

Privacy in machine learning is the ML system's ability to preserve private data information. For the formal definition, we use the most popular *differential privacy* taken from the work of Dwork [58].

Definition 9 (ϵ-Differential Privacy). *Let $\mathcal{A}$ be a randomised algorithm. Let D_1 and D_2 be two training data sets that differ only on one instance. Let S be a subset of the output set of $\mathcal{A}$. $\mathcal{A}$ gives ϵ-differential privacy if*

$$Pr[\mathcal{A}(D_1) \in S] \leq exp(\epsilon) * Pr[\mathcal{A}(D_2) \in S]. \tag{7}$$

In other words, ϵ-Differential privacy is a form of ϵ-contained bound on output change in responding to single input change. It provides a way to know whether any one individual's data has has a significant effect (bounded by ϵ) on the outcome.

Data privacy has been regulated by law makers, for example, the EU General Data Protection Regulation (GDPR) [59] and California Consumer Privacy Act (CCPA) [60]. Current research mainly focuses on how to present privacy-preserving machine learning, instead of detecting privacy violations. We discuss privacy-related research opportunities and research directions in Section 10.

3.4.6 Efficiency

The efficiency of a machine learning system refers to its construction or prediction speed. An efficiency problem happens when the system executes slowly or even infinitely during the construction or the prediction phase.

With the exponential growth of data and complexity of systems, efficiency is an important feature to consider for model selection and framework selection, sometimes even more important than accuracy [61]. For example, to deploy a large model to a mobile device, optimisation, compression, and device-oriented customisation may be performed to make it feasible for the mobile device execution in a reasonable time, but accuracy may sacrifice to achieve this.

3.4.7 Fairness

Machine learning is a statistical method and is widely adopted to make decisions, such as income prediction and medical treatment prediction. Machine learning tends to learn what humans teach it (i.e., in form of training data). However, humans may have bias over cognition, further affecting the data collected or labelled and the algorithm designed, leading to bias problems.

The characteristics that are sensitive and need to be protected against unfairness are called *protected characteristics* [62] or *protected attributes* and *sensitive attributes*. Examples of legally recognised protected classes include race, colour, sex, religion, national origin, citizenship, age, pregnancy, familial status, disability status, veteran status, and genetic information.

Fairness is often domain specific. Regulated domains include credit, education, employment, housing, and public accommodation.[4]

To formulate fairness is the first step to solve the fairness problems and build fair machine learning models. The literature has proposed many definitions of fairness but no firm consensus is reached at this moment. Considering that the definitions themselves are the research focus of fairness in machine learning, we discuss how the literature formulates and measures different types of fairness in Section 6.5.

4. To prohibit discrimination 'in a place of public accommodation on the basis of sexual orientation, gender identity, or gender expression' [63].

TABLE 1
Comparison Between Traditional Software Testing and ML Testing

Characteristics	Traditional Testing	ML Testing
Component to test	code	data and code (learning program, framework)
Behaviour under test	usually fixed	change overtime
Test input	input data	data or code
Test oracle	defined by developers	defined by developers and labelling companies
Adequacy criteria	coverage/mutation score	unknown
False positives in bugs	rare	prevalent
Tester	developer	data scientist, algorithm designer, developer

3.4.8 Interpretability

Machine learning models are often applied to assist/make decisions in medical treatment, income prediction, or personal credit assessment. It may be important for humans to understand the 'logic' behind the final decisions, so that they can build trust over the decisions made by ML [64], [65], [66].

The motives and definitions of interpretability are diverse and still somewhat discordant [64]. Nevertheless, unlike fairness, a mathematical definition of ML interpretability remains elusive [65]. Referring to the work of Biran and Cotton [67] as well as the work of Miller [68], we describe the interpretability of ML as the degree to which an observer can understand the cause of a decision made by an ML system.

Interpretability contains two aspects: transparency (how the model works) and post hoc explanations (other information that could be derived from the model) [64]. Interpretability is also regarded as a request by regulations like GDPR [69], where the user has the legal 'right to explanation' to ask for an explanation of an algorithmic decision that was made about them. A thorough introduction of ML interpretability can be referred to in the book of Christoph [70].

3.5 Software Testing versus ML Testing

Traditional software testing and ML testing are different in many aspects. To understand the unique features of ML testing, we summarise the primary differences between traditional software testing and ML testing in Table 1.

1) *Component to test* (where the bug may exist): traditional software testing detects bugs in the code, while ML testing detects bugs in the data, the learning program, and the framework, each of which play an essential role in building an ML model.
2) *Behaviours under test*: the behaviours of traditional software code are usually fixed once the requirement is fixed, while the behaviours of an ML model may frequently change as the training data is updated.
3) *Test input*: the test inputs in traditional software testing are usually the input data when testing code; in ML testing, however, the test inputs in may have more diverse forms. Note that we separate the definition of 'test input' and 'test data'. In particular, we use 'test input' to refer to the inputs in any form that can be adopted to conduct machine learning testing; while 'test data' specially refers to the data used to validate ML model behaviour (see more in Section 2). Thus, test inputs in ML testing could be, but are not limited to, test data. When testing the learning program, a test case may be a single test instance from the test data or a toy training set; when testing the data, the test input could be a learning program.
4) *Test oracle*: traditional software testing usually assumes the presence of a test oracle. The output can be verified against the expected values by the developer, and thus the oracle is usually determined beforehand. Machine learning, however, is used to generate answers based on a set of input values after being deployed online. The correctness of the large number of generated answers is typically manually-confirmed. Currently, the identification of test oracles remains challenging, because many desired properties are difficult to formally specify. Even for a concrete domain specific problem, the oracle identification is still time-consuming and labour-intensive, because domain-specific knowledge is often required. In current practices, companies usually rely on third-party data labelling companies to get manual labels, which can be expensive.Metamorphic relations [71] are a type of pseudo oracle adopted to automatically mitigate the oracle problem in machine learning testing.
5) *Test adequacy criteria*: test adequacy criteria are used to provide quantitative measurement on the degree of the target software that has been tested. Up to present, many adequacy criteria are proposed and widely adopted in industry, e.g., line coverage, branch coverage, dataflow coverage. However, due to fundamental differences of programming paradigm and logic representation format for machine learning software and traditional software, new test adequacy criteria are required to take the characteristics of machine learning software into consideration.
6) *False positives in detected bugs*: due to the difficulty in obtaining reliable oracles, ML testing tends to yield more false positives in the reported bugs.
7) *Roles of testers*: the bugs in ML testing may exist not only in the learning program, but also in the data or the algorithm, and thus data scientists or algorithm designers could also play the role of testers.

4 Paper Collection and Review Schema

This section introduces the scope, the paper collection approach, an initial analysis of the collected papers, and the organisation of our survey.

4.1 Survey Scope

An ML system may include both hardware and software. The scope of our paper is software testing (as defined in the introduction) applied to machine learning.

We apply the following inclusion criteria when collecting papers. If a paper satisfies any one or more of the following criteria, we will include it. When speaking of related 'aspects of ML testing', we refer to the ML properties, ML components, and ML testing procedure introduced in Section 2.

1) The paper introduces/discusses the general idea of ML testing or one of the related aspects of ML testing.
2) The paper proposes an approach, study, or tool/framework that targets testing one of the ML properties or components.
3) The paper presents a dataset or benchmark especially designed for the purpose of ML testing.
4) The paper introduces a set of measurement criteria that could be adopted to test one of the ML properties.

Some papers concern traditional validation of ML model performance such as the introduction of precision, recall, and cross-validation. We do not include these papers because they have had a long research history and have been thoroughly and maturely studied. Nevertheless, for completeness, we include the knowledge when introducing the background to set the context. We do not include the papers that adopt machine learning techniques for the purpose of traditional software testing and also those target ML problems, which do not use testing techniques as a solution.

Some recent papers also target the formal guarantee on the desired properties of a machine learning system, i.e., to formally verify the correctness of the machine learning systems as well as other properties. Testing and verification of machine learning, as in traditional testing, have their own advantages and disadvantages. For example, verification usually requires a white-box scenario, but suffers from poor scalability, while testing may scale, but lacks completeness. The size of the space of potential behaviours may render current approaches to verification infeasible in general [72], but specific safety critical properties will clearly benefit from focused research activity on scalable verification, as well as testing. In this survey, we focus on the machine learning testing. More details for the literature review of the verification of machine learning systems can be found in the recent work of Xiang *et al* . [73].

4.2 Paper Collection Methodology

To collect the papers across different research areas as much as possible, we started by using exact keyword searching on popular scientific databases including Google Scholar, DBLP and arXiv one by one. The keywords used for searching are listed below. [ML properties] means the set of ML testing properties including correctness, model relevance, robustness, efficiency, privacy, fairness, and interpretability. We used each element in this set plus 'test' or 'bug' as the search query. Similarly, [ML components] denotes the set of ML components including data, learning program/code, and framework/library. Altogether, we conducted $(3 * 3 + 6 * 2 + 3 * 2) * 3 = 81$ searches across the three repositories before May 15th, 2019.

- machine learning + test | bug | trustworthiness
- deep learning + test | bug | trustworthiness
- neural network + test | bug | trustworthiness
- [ML properties]+ test | bug
- [ML components]+ test | bug

TABLE 2
Paper Query Results

Key Words	Hits	Title	Body
machine learning test	211	17	13
machine learning bug	28	4	4
machine learning trustworthiness	1	0	0
deep learning test	38	9	8
deep learning bug	14	1	1
deep learning trustworthiness	2	1	1
neural network test	288	10	9
neural network bug	22	0	0
neural network trustworthiness	5	1	1
[ML properties]+test	294	5	5
[ML properties]+bug	10	0	0
[ML components]+test	77	5	5
[ML components]+bug	8	2	2
Query	-	-	50
Snowball	-	-	59
Author feedback	-	-	35
Overall	-	-	144

Machine learning techniques have been applied in various domains across different research areas. As a result, authors may tend to use very diverse terms. To ensure a high coverage of ML testing related papers, we therefore also performed snowballing [74] on each of the related papers found by keyword searching. We checked the related work sections in these studies and continue adding the related work that satisfies the inclusion criteria introduced in Section 4.1, until we reached closure.

To ensure a more comprehensive and accurate survey, we emailed the authors of the papers that were collected via query and snowballing, and let them send us other papers they are aware of which are related with machine learning testing but have not been included yet. We also asked them to check whether our description about their work in the survey was accurate and correct.

4.3 Collection Results

Table 2 shows the details of paper collection results. The papers collected from Google Scholar and arXiv turned out to be subsets of those from DBLP so we only present the results of DBLP. Keyword search and snowballing resulted in 109 papers across six research areas till May 15th, 2019. We received over 50 replies from all the cited authors until June 4th, 2019, and added another 35 papers when dealing with the author feedback. Altogether, we collected 144 papers.

Fig. 7 shows the distribution of papers published in different research venues. Among all the papers, 38.2 percent papers are published in software engineering venues such as ICSE, FSE, ASE, ICST, and ISSTA; 6.9 percent papers are published in systems and network venues; surprisingly, only 19.4 percent of the total papers are published in artificial intelligence venues such as AAAI, CVPR, and ICLR. Additionally, 22.9 percent of the papers have not yet been published via peer-reviewed venues (the arXiv part).

4.4 Paper Organisation

We present the literature review from two aspects: 1) a literature review of the collected papers, 2) a statistical analysis

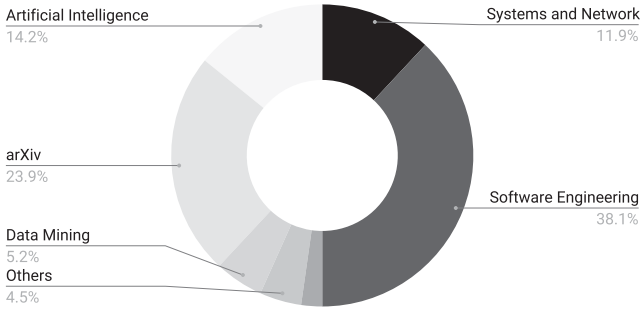


Fig. 7. Publication venue distribution.

of the collected papers, datasets, and tools. The sections and the corresponding contents are presented in Table 3.

1) Literature Review. The papers in our collection are organised and presented from four angles. We introduce the work about different testing workflow in Section 5. In Section 6 we classify the papers based on the ML problems they target, including functional properties like correctness and model relevance and non-functional properties like robustness, fairness, privacy, and interpretability. Section 7 introduces the testing technologies on detecting bugs in data, learning programs, and ML frameworks, libraries, or platforms. Section 8 introduces the testing techniques applied in particular application scenarios such as autonomous driving and machine translation.

The four aspects have different focuses of ML testing, each of which is a complete organisation of the total collected papers (see more discussion in Section 3.1), as a single ML testing paper may fit multiple aspects if being viewed from different angles.

2) Statistical Analysis and Summary. We analyse and compare the number of research papers on different machine learning categories (supervised/unsupervised/reinforcement learning), machine learning structures (classic/deep learning), testing properties in Section 9. We also summarise the datasets and tools adopted so far in ML testing.

The four different angles for presentation of related work as well as the statistical summary, analysis, and comparison, enable us to observe the research focus, trend, challenges, opportunities, and directions of ML testing. These results are presented in Section 10.

5 ML Testing Workflow

This section organises ML testing research based on the testing workflow as shown by Fig. 5.

ML testing includes offline testing and online testing. Albarghouthi and Vinitsky [75] developed a fairness specification language that can be used for the development of run-time monitoring, in detecting fairness issues. Such a kind of run-time monitoring belongs to the area of online testing. Nevertheless, current research mainly centres on offline testing as introduced below. The procedures that are not covered based on our paper collection, such as requirement analysis and regression testing and those belonging to online testing are discussed as research opportunities in Section 10.

5.1 Test Input Generation

We organise the test input generation research based on the techniques adopted.

TABLE 3
Review Schema

Classification	Sec	Topic
Testing Workflow	Section 5.1	Test Input Generation
	Section 5.2	Test Oracle
	Section 5.3	Test Adequacy
	Section 5.4	Test Prioritisation and Reduction
	Section 5.5	Bug Report Analysis
	Section 5.6	Debug and Repair
	Section 5.7	Testing Framework and Tools
Testing Properties	Section 6.1	Correctness
	Section 6.2	Model Revelance
	Section 6.3	Robustness and Security
	Section 6.4	Efficiency
	Section 6.5	Fairness
	Section 6.6	Interpretability
	Section 6.7	Privacy
Testing Components	Section 7.1	Bug Detection in Data
	Section 7.2	Bug Detection in Learning Program
	Section 7.3	Bug Detection in Framework
Application Scenario	Section 8.1	Autonomous Driving
	Section 8.2	Machine Translation
	Section 8.3	Natural Language Inference
Summary & Analysis	Section 9.1	Timeline
	Section 9.2	Research Distribution among Categories
	Section 9.3	Research Distribution among Properties
	Section 9.4	Datasets
	Section 9.5	Open-source Tool Support

5.1.1 Domain-Specific Test Input Synthesis

Test inputs of ML testing can be classified into two categories: adversarial inputs and natural inputs. Adversarial inputs are perturbed based on the original inputs. They may not belong to normal data distribution (i.e., maybe rarely exist in practice), but could expose robustness or security flaws. Natural inputs, instead, are those inputs that belong to the data distribution of a practical application scenario. Here we introduce the related work that aims to generate natural inputs via domain-specific test input synthesis.

DeepXplore [1] proposed a white-box differential testing technique to generate test inputs for a deep learning system. Inspired by test coverage in traditional software testing, the authors proposed neuron coverage to drive test generation (we discuss different coverage criteria for ML testing in Section 5.2.3). The test inputs are expected to have high neuron coverage. Additionally, the inputs need to expose differences among different DNN models, as well as be like real-world data as much as possible. The joint optimisation algorithm iteratively uses a gradient search to find a modified input that satisfies all of these goals. The evaluation of DeepXplore indicates that it covers 34.4 and 33.2 percent more neurons than the same number of randomly picked inputs and adversarial inputs.

To create useful and effective data for autonomous driving systems, *DeepTest* [76] performed greedy search with nine different realistic image transformations: changing brightness, changing contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect. There are three types of image transformation styles provided in OpenCV:[5] linear, affine, and convolutional. The evaluation of DeepTest uses the Udacity self-driving car challenge dataset [77]. It detected more than 1,000 erroneous behaviours on CNNs and RNNs with low false positive rates.[6]

Generative adversarial networks (GANs) [78] are algorithms to generate models that approximate the manifolds and distribution on a given set of data. GAN has been successfully applied to advanced image transformation (e.g., style transformation, scene transformation) that look at least superficially authentic to human observers. Zhang *et al.* [79] applied GAN to deliver driving scene-based test generation with various weather conditions. They sampled images from Udacity Challenge dataset [77] and YouTube videos (snowy or rainy scenes), and fed them into the UNIT framework[7] for training. The trained model takes the whole Udacity images as the seed inputs and yields transformed images as generated tests.

Zhou *et al.* [81] proposed *DeepBillboard* to generate real-world adversarial billboards that can trigger potential steering errors of autonomous driving systems.

To test audio-based deep learning systems, Du *et al.* [82] designed a set of transformations tailored to audio inputs considering background noise and volume variation. They first abstracted and extracted a probabilistic transition model from an RNN. Based on this, stateful testing criteria are defined and used to guide test generation for stateful machine learning system.

To test the image classification platform when classifying biological cell images, Ding *et al.* [83] built a testing framework for biological cell classifiers. The framework iteratively generates new images and uses metamorphic relations for testing. For example, they generate new images by increasing the number/shape of artificial mitochondrion into the biological cell images, which can arouse easy-to-identify changes in the classification results.

Rabin *et al.* [84] discussed the possibilities of testing code2vec (a code embedding approach [85]) with semantic-preserving program transformations serving as test inputs.

To test machine translation systems, Sun *et al.* [86] automatically generate test inputs via mutating the words in translation inputs. In order to generate translation pairs that ought to yield consistent translations, their approach conducts word replacement based on word embedding similarities. Manual inspection indicates that the test generation has a high precision (99 percent) on generating input pairs with consistent translations.

5.1.2 Fuzz and Search-Based Test Input Generation

Fuzz testing is a traditional automatic testing technique that generates random data as program inputs to detect crashes, memory leaks, failed (built-in) assertions, etc, with many sucessfully application to system security and vulnerability detection [9]. As another widely used test generation technique, search-based test generation often uses metaheuristic search techniques to guide the fuzz process for more efficient and effective test generation [17], [87], [88]. These two techniques have also been proved to be effective in exploring the input space of ML testing:

Odena *et al.* [89] presented *TensorFuzz*. *TensorFuzz* used a simple nearest neighbour hill climbing approach to explore achievable coverage over valid input space for Tensorflow graphs, and to discover numerical errors, disagreements between neural networks and their quantized versions, and surfacing undesirable behaviour in RNNs.

DLFuzz, proposed by Guo *et al.* [90], is another fuzz test generation tool based on the implementation of DeepXplore with nueron coverage as guidance. DLFuzz aims to generate adversarial examples. The generation process thus does not require similar functional deep learning systems for cross-referencing check like DeepXplore and TensorFuzz. Rather, it needs only minimum changes over the original inputs to find those new inputs that improve neural coverage but have different predictive results from the original inputs. The preliminary evaluation on MNIST and ImageNet shows that compared with DeepXplore, DLFuzz is able to generate 135 to 584.62 percent more inputs with 20.11 percent less time consumption.

Xie *et al.* [91] presented a metamorphic transformation based coverage guided fuzzing technique, DeepHunter, which leverages both neuron coverage and coverage criteria presented by DeepGauge [92]. DeepHunter uses a more fine-grained metamorphic mutation strategy to generate tests, which demonstrates the advantage in reducing the false positive rate. It also demonstrates its advantage in achieving high coverage and bug detection capability.

Wicker *et al.* [93] proposed feature-guided test generation. They adopted Scale-Invariant Feature Transform (SIFT) to identify features that represent an image with a Gaussian mixture model, then transformed the problem of finding adversarial examples into a two-player turn-based stochastic game. They used Monte Carlo Tree Search to identify those elements of an image most vulnerable as the means of generating adversarial examples. The experiments show that their black-box approach is competitive with some state-of-the-art white-box methods.

Instead of targeting supervised learning, Uesato *et al.* [94] proposed to evaluate reinforcement learning with adversarial example generation. The detection of catastrophic failures is expensive because failures are rare. To alleviate the consequent cost of finding such failures, the authors proposed to use a failure probability predictor to estimate the probability that the agent fails, which was demonstrated to be both effective and efficient.

There are also fuzzers for specific application scenarios other than image classifications. Zhou *et al.* [95] combined fuzzing and metamorphic testing to test the LiDAR obstacle-perception module of real-life self-driving cars, and reported previously unknown software faults. Jha *et al.* [96] investigated how to generate the most effective test cases (the faults that are most likely to lead to violations of safety conditions) via modelling the fault injection as a Bayesian network. The

5. https://github.com/itseez/opencv(2015)

6. The examples of detected erroneous behaviours are available at https://deeplearningtest.github.io/deepTest/.

7. A recent DNN-based method to perform image-to-image transformation [80]

evaluation, based on two production-grade AV systems from NVIDIA and Baidu, revealed many situations where faults lead to safety violations.

Udeshi and Chattopadhyay [97] generate inputs for text classification tasks and produce a fuzzing approach that considers the grammar under test as well as the distance between inputs. Nie *et al.* [98] and Wang *et al.* [99] mutated the sentences in Natural Language Inference (NLI) tasks to generate test inputs for robustness testing. Chan *et al.* [100] generated adversarial examples for DNC to expose its robustness problems. Udeshi *et al.* [101] focused much on individual fairness and generated test inputs that highlight the discriminatory nature of the model under test. We give details about these domain-specific fuzz testing techniques in Section 8.

Tuncali *et al.* [102] proposed a framework for testing autonomous driving systems. In their work they compared three test generation strategies: random fuzz test generation, covering array [103] + fuzz test generation, and covering array + search-based test generation (using Simulated Annealing algorithm [104]). The results indicated that the test generation strategy with search-based technique involved has the best performance in detecting glancing behaviours.

5.1.3 Symbolic Execution Based Test Input Generation

Symbolic execution is a program analysis technique to test whether certain properties can be violated by the software under test [105]. Dynamic Symbolic Execution (DSE, also called concolic testing) is a technique used to automatically generate test inputs that achieve high code coverage. DSE executes the program under test with random test inputs and performs symbolic execution in parallel to collect symbolic constraints obtained from predicates in branch statements along the execution traces. The conjunction of all symbolic constraints along a path is called a path condition. When generating tests, DSE randomly chooses one test input from the input domain, then uses constraint solving to reach a target branch condition in the path [106]. DSE has been found to be accurate and effective, and has been the primary technique used by some vulnerability discovery tools [107].

In ML testing, the model's performance is decided, not only by the code, but also by the data, and thus symbolic execution has two application scenarios: either on the data or on the code.

Symbolic analysis was applied to generate more effective tests to expose bugs by Ramanathan and Pullum [7]. They proposed a combination of symbolic and statistical approaches to efficiently find test cases. The idea is to distance-theoretically abstract the data using symbols to help search for those test inputs where minor changes in the input will cause the algorithm to fail. The evaluation of the implementation of a k-means algorithm indicates that the approach is able to detect subtle errors such as bit-flips. The examination of false positives may also be a future research interest.

When applying symbolic execution on the machine learning code, there are many challenges. Gopinath [108] listed three such challenges for neural networks in their paper, which work for other ML modes as well: (1) the networks have no explicit branching; (2) the networks may be highly non-linear, with no well-developed solvers for constraints; and (3) there are scalability issues because the structures of the ML models are usually very complex and are beyond the capabilities of current symbolic reasoning tools.

Considering these challenges, Gopinath [108] introduced DeepCheck. It transforms a Deep Neural Network (DNN) into a program to enable symbolic execution to find pixel attacks that have the same activation pattern as the original image. In particular, the activation functions in DNN follow an IF-Else branch structure, which can be viewed as a path through the translated program. DeepCheck is able to create 1-pixel and 2-pixel attacks by identifying most of the pixels or pixel-pairs that the neural network fails to classify the corresponding modified images.

Similarly, Agarwal *et al.* [109] apply LIME [110], a local explanation tool that approximates a model with linear models, decision trees, or falling rule lists, to help get the path used in symbolic execution. Their evaluation based on 8 open source fairness benchmarks shows that the algorithm generates 3.72 times more successful test cases than the random test generation approach THEMIS [5].

Sun *et al.* [111] presented DeepConcolic, a dynamic symbolic execution testing method for DNNs. Concrete execution is used to direct the symbolic analysis to particular MC/DC criteria' condition, through concretely evaluating given properties of the ML models. DeepConcolic explicitly takes coverage requirements as input. The authors report that it yields over 10 percent higher neuron coverage than DeepXplore for the evaluated models.

5.1.4 Synthetic Data to Test Learning Program

Murphy *et al.* [112] generated data with repeating values, missing values, or categorical data for testing two ML ranking applications. Breck *et al.* [45] used synthetic training data that adhere to schema constraints to trigger the hidden assumptions in the code that do not agree with the constraints. Zhang *et al.* [54] used synthetic data with known distributions to test overfitting. Nakajima and Bui [113] also mentioned the possibility of generating simple datasets with some predictable characteristics that can be adopted as pseudo oracles.

5.2 Test Oracle

Test oracle identification is one of the key problems in ML testing. It is needed in order to enable the judgement of whether a bug exists. This is the so-called 'Oracle Problem' [9].

In ML testing, the oracle problem is challenging, because many machine learning algorithms are probabilistic programs. In this section, we list several popular types of test oracle that have been studied for ML testing, i.e., metamorphic relations, cross-referencing, and model evaluation metrics.

5.2.1 Metamorphic Relations as Test Oracles

Metamorphic relations was proposed by Chen *et al.* [114] to ameliorate the test oracle problem in traditional software testing. A metamorphic relation refers to the relationship between the software input change and output change during multiple program executions. For example, to test the implementation of the function $sin(x)$, one may check how the function output changes when the input is changed from x to $\pi - x$. If $sin(x)$ differs from $sin(\pi - x)$, this observation signals an error without needing to examine the

specific values computed by the implementation. $sin(x) = sin(\pi - x)$ is thus a metamorphic relation that plays the role of test oracle (also named 'pseudo oracle') to help bug detection.

In ML testing, metamorphic relations are widely studied to tackle the oracle problem. Many metamorphic relations are based on transformations of training or test data that are expected to yield unchanged or certain expected changes in the predictive output. There are different granularities of data transformations when studying the corresponding metamorphic relations. Some transformations conduct coarse-grained changes such as enlarging the dataset or changing the data order, without changing each single data instance. We call these transformations 'Coarse-grained data transformations'. Some transformations conduct data transformations via smaller changes on each data instance, such as mutating the attributes, labels, or pixels of images, and are referred to as 'fine-grained' data transformations in this paper. The related works of each type of transformations are introduced below.

Coarse-Grained Data Transformation. As early as in 2008, Murphy *et al.* [115] discuss the properties of machine learning algorithms that may be adopted as metamorphic relations. Six transformations of input data are introduced: additive, multiplicative, permutative, invertive, inclusive, and exclusive. The changes include adding a constant to numerical values; multiplying numerical values by a constant; permuting the order of the input data; reversing the order of the input data; removing a part of the input data; adding additional data. Their analysis is on MartiRank, SVM-Light, and PAYL. Although unevaluated in the initial 2008 paper, this work provided a foundation for determining the relationships and transformations that can be used for conducting metamorphic testing for machine learning.

Ding *et al.* [116] proposed 11 metamorphic relations to test deep learning systems. At the dataset level, the metamorphic relations were also based on training data or test data transformations that were not supposed to affect classification accuracy, such as adding 10 percent training images into each category of the training data set or removing one category of data from the dataset. The evaluation is based on a classification of biological cell images.

Murphy *et al.* [117] presented function-level metamorphic relations. The evaluation on 9 machine learning applications indicated that functional-level properties were 170 percent more effective than application-level properties.

Fine-Grained Data Transformation. In 2009, Xie *et al.* [118] proposed to use metamorphic relations that were specific to a certain model to test the implementations of supervised classifiers. The paper presents five types of metamorphic relations that enable the prediction of expected changes to the output (such as changes in classes, labels, attributes) based on particular changes to the input. Manual analysis of the implementation of KNN and Naive Bayes from Weka [119] indicates that not all metamorphic relations are necessary. The differences in the metamorphic relations between SVM and neural networks are also discussed in [120]. Dwarakanath *et al.* [121] applied metamorphic relations to image classifications with SVM and deep learning systems. The changes on the data include changing the feature or instance orders, linear scaling of the test features, normalisation or scaling up the test data, or changing the convolution operation order of the data. The proposed MRs are able to find 71 percent of the injected bugs. Sharma and Wehrheim [122] considered fine-grained data transformations such as changing feature names, renaming feature values to test fairness. They studied 14 classifiers, none of them were found to be sensitive to feature name shuffling.

Zhang *et al.* [54] proposed Perturbed Model Validation (PMV) which combines metamorphic relation and data mutation to detect overfitting. PMV mutates the training data via injecting noise in the training data to create perturbed training datasets, then checks the training accuracy decrease rate when the noise degree increases. The faster the training accuracy decreases, the less the machine learning model overfits.

Al-Azani and Hassine [123] studied the metamorphic relations of Naive Bayes, k-Nearest Neighbour, as well as their ensemble classifier. It turns out that the metamorphic relations necessary for Naive Bayes and k-Nearest Neighbour may be not necessary for their ensemble classifier.

Tian *et al.* [76] and Zhang *et al.* [79] stated that the autonomous vehicle steering angle should not change significantly or stay the same for the transformed images under different weather conditions. Ramanagopal *et al.* [124] used the classification consistency of similar images to serve as test oracles for testing self-driving cars. The evaluation indicates a precision of 0.94 when detecting errors in unlabelled data.

Additionally, Xie *et al.* [125] proposed METTLE, a metamorphic testing approach for unsupervised learning validation. METTLE has six types of different-grained metamorphic relations that are specially designed for unsupervised learners. These metamorphic relations manipulate instance order, distinctness, density, attributes, or inject outliers of the data. The evaluation was based on synthetic data generated by Scikit-learn, showing that METTLE is practical and effective in validating unsupervised learners. Nakajima *et al.* [113], [126] discussed the possibilities of using different-grained metamorphic relations to find problems in SVM and neural networks, such as to manipulate instance order or attribute order and to reverse labels and change attribute values, or to manipulate the pixels in images.

Metamorphic Relations Between Different Datasets. The consistency relations between/among different datasets can also be regarded as metamorphic relations that could be applied to detect data bugs. Kim *et al.* [127] and Breck *et al.* [45] studied the metamorphic relations between training data and new data. If the training data and new data have different distributions, the training data may not be adequate. Breck *et al.* [45] also studied the metamorphic relations among different datasets that are close in time: these datasets are expected to share some characteristics because it is uncommon to have frequent drastic changes to the data-generation code.

Frameworks to Apply Metamorphic Relations. Murphy *et al.* [128] implemented a framework called Amsterdam to automate the process of using metamorphic relations to detect ML bugs. The framework reduces false positives via setting thresholds when doing result comparison. They also developed Corduroy [117], which extended Java Modelling Language to let developers specify metamorphic properties and generate test cases for ML testing.

We introduce more related work on domain-specific metamorphic relations of testing autonomous driving, Differentiable Neural Computer (DNC) [100], machine translation systems [86], [129], [130], biological cell classification [83], and audio-based deep learning systems [82] in Section 8.

5.2.2 Cross-Referencing as Test Oracles

Cross-Referencing is another type of test oracle for ML testing, including differential Testing and N-version Programming. Differential testing is a traditional software testing technique that detects bugs by observing whether similar applications yield different outputs regarding identical inputs [11], [131]. It is a testing oracle for detecting compiler bugs [132]. According to the study of Nejadgholi and Yang [133], 5 to 27 percent test oracles for deep learning libraries use differential testing.

Differential testing is closely related with N-version programming [134]: N-version programming aims to generate multiple functionally-equivalent programs based on one specification, so that the combination of different versions are more fault-tolerant and robust.

Davis and Weyuker [11] discussed the possibilities of differential testing for 'non-testable' programs. The idea is that if multiple implementations of an algorithm yield different outputs on one identical input, then at least one of the implementation contains a defect. Alebiosu *et al.* [135] evaluated this idea on machine learning, and successfully found 16 faults from 7 Naive Bayes implementations and 13 faults from 19 k-nearest neighbour implementation.

Pham *et al.* [48] also adopted cross referencing to test ML implementations, but focused on the implementation of deep learning libraries. They proposed CRADLE, the first approach that focuses on finding and localising bugs in deep learning software libraries. The evaluation was conducted on three libraries (TensorFlow, CNTK, and Theano), 11 datasets (including ImageNet, MNIST, and KGS Go game), and 30 pre-trained models. It turned out that CRADLE detects 104 unique inconsistencies and 12 bugs.

DeepXplore [1] and DLFuzz [90] used differential testing as test oracles to find effective test inputs. Those test inputs causing different behaviours among different algorithms or models were preferred during test generation.

Most differential testing relies on multiple implementations or versions, while Qin *et al.* [136] used the behaviours of 'mirror' programs, generated from the training data as pseudo oracles. A mirror program is a program generated based on training data, so that the behaviours of the program represent the training data. If the mirror program has similar behaviours on test data, it is an indication that the behaviour extracted from the training data suit test data as well.

Sun *et al.* [86] applied cross reference in repairing machine translation systems. Their approach, TransRepair, compares the outputs (i.e., translations) of different mutated inputs, and picks the output that shares the most similarity with others as a superior translation candidate.

5.2.3 Measurement Metrics for Designing Test Oracles

Some work has presented definitions or statistical measurements of non-functional features of ML systems including robustness [137], fairness [138], [139], [140], and interpretability [65], [141]. These measurements are not direct oracles for testing, but are essential for testers to understand and evaluate the property under test, and to provide some actual statistics that can be compared with the expected ones. For example, the definitions of different types of fairness [138], [139], [140] (more details are in Section 6.5.1) define the conditions an ML system has to satisfy without which the system is not fair. These definitions can be adopted directly to detect fairness violations.

Except for these popular test oracles in ML testing, there are also some domain-specific rules that could be applied to design test oracles. We discussed several domain-specific rules that could be adopted as oracles to detect data bugs in Section 7.1.1. Kang *et al.* [142] discussed two types of model assertions under the task of car detection in videos: flickering assertion to detect the flickering in car bounding box, and multi-box assertion to detect nested-car bounding. For example, if a car bounding box contains other boxes, the multi-box assertion fails. They also proposed some automatic fix rules to set a new predictive result when a test assertion fails.

There has also been a discussion about the possibility of evaluating ML learning curve in lifelong machine learning as the oracle [143]. An ML system can pass the test oracle if it can grow and increase its knowledge level over time.

5.3 Test Adequacy

Test adequacy evaluation aims to discover whether the existing tests have a good fault-revealing ability. It provides an objective confidence measurement on testing activities. The adequacy criteria can also be adopted to guide test generation. Popular test adequacy evaluation techniques in traditional software testing include code coverage and mutation testing, which are also adopted in ML testing.

5.3.1 Test Coverage

In traditional software testing, code coverage measures the degree to which the source code of a program is executed by a test suite [144]. The higher coverage a test suite achieves, it is more probable that the hidden bugs could be uncovered. In other words, covering the code fragment is a necessary condition to detect the defects hidden in the code. It is often desirable to create test suites to achieve higher coverage.

Unlike traditional software, code coverage is seldom a demanding criterion for ML testing, since the decision logic of an ML model is not written manually but rather it is learned from training data. For example, in the study of Pei *et al.* [1], 100 percent traditional code coverage is easy to achieve with a single randomly chosen test input. Instead, researchers propose various types of coverage for ML models beyond code coverage.

Neuron Coverage. Pei *et al.* [1] proposed the first coverage criterion, neuron coverage, particularly designed for deep learning testing. Neuron coverage is calculated as the ratio of the number of unique neurons activated by all test inputs and the total number of neurons in a DNN. In particular, a neuron is activated if its output value is larger than a user-specified threshold.

Ma *et al.* [92] extended the concept of neuron coverage. They first profile a DNN based on the training data, so that they obtain the activation behaviour of each neuron against

the training data. Based on this, they propose more fine-grained criteria, *k*-multisection neuron coverage, neuron boundary coverage, and strong neuron activation coverage, to represent the major functional behaviour and corner behaviour of a DNN.

MC/DC Coverage Variants. Sun *et al.* [145] proposed four test coverage criteria that are tailored to the distinct features of DNN inspired by the MC/DC coverage criteria [146]. MC/DC observes the change of a Boolean variable, while their proposed criteria observe a sign, value, or distance change of a neuron, in order to capture the causal changes in the test inputs. The approach assumes the DNN to be a fully-connected network, and does not consider the context of a neuron in its own layer as well as different neuron combinations within the same layer [147].

Layer-Level Coverage. Ma *et al.* [92] also presented layer-level coverage criteria, which considers the top hyperactive neurons and their combinations (or the sequences) to characterise the behaviours of a DNN. The coverage is evaluated to have better performance together with neuron coverage based on dataset MNIST and ImageNet. In their following-up work [148], [149], they further proposed combinatorial testing coverage, which checks the combinatorial activation status of the neurons in each layer via checking the fraction of neurons activation interaction in a layer. Sekhon and Fleming [147] defined a coverage criteria that looks for 1) all pairs of neurons in the same layer having all possible value combinations, and 2) all pairs of neurons in consecutive layers having all possible value combinations.

State-Level Coverage. While aftermentioned criteria, to some extent, capture the behaviours of feed-forward neural networks, they do not explicitly characterise stateful machine learning system like recurrent neural network (RNN). The RNN-based ML approach has achieved notable success in applications that handle sequential inputs, e.g., speech audio, natural language, cyber physical control signals. In order to analyse such stateful ML systems, Du *et al.* [82] proposed the first set of testing criteria specialised for RNN-based stateful deep learning systems. They first abstracted a stateful deep learning system as a probabilistic transition system. Based on the modelling, they proposed criteria based on the state and traces of the transition system, to capture the dynamic state transition behaviours.

Limitations of Coverage Criteria. Although there are different types of coverage criteria, most of them focus on DNNs. Sekhon and Fleming [147] examined the existing testing methods for DNNs and discussed the limitations of these criteria.

Most proposed coverage criteria are based on the structure of a DNN. Li *et al.* [150] pointed out the limitations of structural coverage criteria for deep networks caused by the fundamental differences between neural networks and human-written programs. Their initial experiments with natural inputs found no strong correlation between the number of misclassified inputs in a test set and its structural coverage. Due to the black-box nature of a machine learning system, it is not clear how such criteria directly relate to the system's decision logic.

5.3.2 Mutation Testing

In traditional software testing, mutation testing evaluates the fault-revealing ability of a test suite via injecting faults [144], [151]. The ratio of detected faults against all injected faults is called the *Mutation Score*.

In ML testing, the behaviour of an ML system depends on, not only the learning code, but also data and model structure. Ma *et al.* [152] proposed DeepMutation, which mutates DNNs at the source level or model level, to make minor perturbation on the decision boundary of a DNN. Based on this, a mutation score is defined as the ratio of test instances of which results are changed against the total number of instances.

Shen *et al.* [153] proposed five mutation operators for DNNs and evaluated properties of mutation on the MINST dataset. They pointed out that domain-specific mutation operators are needed to enhance mutation analysis.

Compared to structural coverage criteria, mutation testing based criteria are more directly relevant to the decision boundary of a DNN. For example, an input data that is near the decision boundary of a DNN, could more easily detect the inconsistency between a DNN and its mutants.

5.3.3 Surprise Adequacy

Kim *et al.* [127] introduced *surprise adequacy* to measure the coverage of discretised input surprise range for deep learning systems. They argued that test diversity is more meaningful when being measured with respect to the training data. A 'good' test input should be 'sufficiently but not overly surprising' comparing with the training data. Two measurements of surprise were introduced: one is based on Keneral Density Estimation (KDE) to approximate the likelihood of the system having seen a similar input during training, the other is based on the distance between vectors representing the neuron activation traces of the given input and the training data (e.g., euclidean distance). These criteria can be adopted to detect adversarial examples. Further investigation is required to determine whether such criteria enable the behaviour boundaries of ML models to be approximated in terms of surprise. It will also be interesting for future work to study the relationship between adversarial examples, natural error samples, and surprise-based criteria.

5.3.4 Rule-Based Checking of Test Adequacy

To ensure the functionality of an ML system, there may be some 'typical' rules that are necessary. Breck *et al.* [154] offered 28 test aspects to consider and a scoring system used by Google. Their focus is to measure how well a given machine learning system is tested. The 28 test aspects are classified into four types: 1) the tests for the ML model itself, 2) the tests for ML infrastructure used to build the model, 3) the tests for ML data used to build the model, and 4) the tests that check whether the ML system works correctly over time. Most of them are some must-to-check rules that could be applied to guide test generation. For example, the training process should be reproducible; all features should be beneficial; there should be no other model that is simpler but better in performance than the current one. Their research indicates that, although ML testing is complex, there are shared properties to design some basic test cases to test the fundamental functionality of the ML system.

5.4 Test Prioritisation and Reduction

Test input generation in ML has a very large input space to cover. On the other hand, we need to label every test instance

so as to judge predictive accuracy. These two aspects lead to high test generation costs. Byun *et al.* [155] used DNN metrics like cross entropy, surprisal, and Bayesian uncertainty to prioritise test inputs. They experimentally showed that these are good indicators of inputs that expose unacceptable behaviours, which are also useful for retraining.

Generating test inputs is also computationally expensive. Zhang *et al.* [156] proposed to reduce costs by identifying those test instances that denote the more effective adversarial examples. The approach is a test prioritisation technique that ranks the test instances based on their sensitivity to noise, because the instances that are more sensitive to noise is more likely to yield adversarial examples.

Li et. al [157] focused on test data reduction in operational DNN testing. They proposed a sampling technique guided by the neurons in the last hidden layer of a DNN, using a cross-entropy minimisation based distribution approximation technique. The evaluation was conducted on pre-trained models with three image datasets: MNIST, Udacity challenge, and ImageNet. The results show that, compared to random sampling, their approach samples only half the test inputs, yet it achieves a similar level of precision.

Ma *et al.* [158] proposed a set of test selection metrics based on the notion of model confidence. Test inputs that are more uncertain to the models are preferred, because they are more informative and should be used to improve the model if being included during retraining. The evaluation shows that their test selection approach has 80 percent more gain than random selection.

5.5 Bug Report Analysis

Thung *et al.* [159] were the first to study machine learning bugs via analysing the bug reports of machine learning systems. 500 bug reports from Apache Mahout, Apache Lucene, and Apache OpenNLP were studied. The explored problems included bug frequency, bug categories, bug severity, and bug resolution characteristics such as bug-fix time, effort, and file number. The results indicated that incorrect implementation counts for the largest proportion of ML bugs, i.e., 22.6 percent of bugs are due to incorrect implementation of defined algorithms. Implementation bugs are also the most severe bugs, and take longer to fix. In addition, 15.6 percent of bugs are non-functional bugs. 5.6 percent of bugs are data bugs.

Zhang *et al.* [160] studied 175 TensorFlow bugs, based on the bug reports from Github or StackOverflow. They studied the symptoms and root causes of the bugs, the existing challenges to detect the bugs and how these bugs are handled. They classified TensorFlow bugs into exceptions or crashes, low correctness, low efficiency, and unknown. The major causes were found to be in algorithm design and implementations such as TensorFlow API misuse (18.9 percent), unaligned tensor (13.7 percent), and incorrect model parameter or structure (21.7 percent)

Banerjee *et al.* [161] analysed the bug reports of autonomous driving systems from 12 autonomous vehicle manufacturers that drove a cumulative total of 1,116,605 miles in California. They used NLP technologies to classify the causes of disengagements into 10 types (A disagreement is a failure that causes the control of the vehicle to switch from the software to the human driver). The issues in machine learning systems and decision control account for the primary cause of 64 percent of all disengagements based on their report analysis.

5.6 Debug and Repair

Data Resampling. The generated test inputs introduced in Section 5.1 only expose ML bugs, but are also studied as a part of the training data and can improve the model's correctness through retraining. For example, DeepXplore achieves up to 3 percent improvement in classification accuracy by retraining a deep learning model on generated inputs. DeepTest [76] improves the model's accuracy by 46 percent.

Ma *et al.* [162] identified the neurons responsible for the misclassification and call them 'faulty neurons'. They resampled training data that influence such faulty neurons to help improve model performance.

Debugging Framework Development. Dutta*et al.* [163] proposed Storm, a program transformation framework to generate smaller programs that can support debugging for machine learning testing. To fix a bug, developers usually need to shrink the program under test to write better bug reports and to facilitate debugging and regression testing. Storm applies program analysis and probabilistic reasoning to simplify probabilistic programs, which helps to pinpoint the issues more easily.

Cai *et al.* [164] presented *tfdbg*, a debugger for ML models built on TensorFlow, containing three key components: 1) the Analyzer, which makes the structure and intermediate state of the runtime graph visible; 2) the NodeStepper, which enables clients to pause, inspect, or modify at a given node of the graph; 3) the RunStepper, which enables clients to take higher level actions between iterations of model training. Vartak *et al.* [165] proposed the MISTIQUE system to capture, store, and query model intermediates to help the debug. Krishnan and Wu [166] presented PALM, a tool that explains a complex model with a two-part surrogate model: a meta-model that partitions the training data and a set of sub-models that approximate the patterns within each partition. PALM helps developers find out the training data that impacts prediction the most, and thereby target the subset of training data that account for the incorrect predictions to assist debugging.

Fix Understanding. Fixing bugs in many machine learning systems is difficult because bugs can occur at multiple points in different components. Nushi *et al.* [167] proposed a human-in-the-loop approach that simulates potential fixes in different components through human computation tasks: humans were asked to simulate improved component states. The improvements of the system are recorded and compared, to provide guidance to designers about how they can best improve the system.

Program Repair. Albarghouthi *et al.* [168] proposed a distribution-guided inductive synthesis approach to repair decision-making programs such as machine learning programs. The purpose is to construct a new program with correct predictive output, but with similar semantics with the original program. Their approach uses sampled instances and the predicted outputs to drive program synthesis in which the program is encoded based on SMT.

5.7 General Testing Framework and Tools

There has also been work focusing on providing a testing tool or framework that helps developers to implement

testing activities in a testing workflow. There is a test framework to generate and validate test inputs for security testing [169]. Dreossi *et al.* [170] presented a CNN testing framework that consists of three main modules: an image generator, a collection of sampling methods, and a suite of visualisation tools. Tramer *et al.* [171] proposed a comprehensive testing tool to help developers to test and debug fairness bugs with an easily interpretable bug report. Nishi *et al.* [172] proposed a testing framework including different evaluation aspects such as allowability, achievability, robustness, avoidability and improvability. They also discussed different levels of ML testing, such as system, software, component, and data testing.

Thomas *et al.* [173] recently proposed a framework for designing machine learning algorithms, which simplifies the regulation of undesired behaviours. The framework is demonstrated to be suitable for regulating regression, classification, and reinforcement algorithms. It allows one to learn from (potentially biased) data while guaranteeing that, with high probability, the model will exhibit no bias when applied to unseen data. The definition of bias is user-specified, allowing for a large family of definitions. For a learning algorithm and a training data, the framework will either returns a model with this guarantee, or a warning that it fails to find such a model with the required guarantee.

6 ML Properties to be Tested

ML properties concern the conditions we should care about for ML testing, and are usually connected with the behaviour of an ML model after training. The poor performance in a property, however, may be due to bugs in any of the ML components (see more in Introduction 7).

This section presents the related work of testing both functional ML properties and non-functional ML properties. Functional properties include correctness (Section 6.1) and overfitting (Section 6.2). Non-functional properties include robustness and security (Section 6.3), efficiency (Section 6.4), fairness (Section 6.5).

6.1 Correctness

Correctness concerns the fundamental function accuracy of an ML system. Classic machine learning validation is the most well-established and widely-used technology for correctness testing. Typical machine learning validation approaches are cross-validation and bootstrap. The principle is to isolate test data via data sampling to check whether the trained model fits new cases. There are several approaches to perform cross-validation. In hold out cross-validation [174], the data are split into two parts: one part becomes the training data and the other part becomes test data.[8] In k-fold cross-validation, the data are split into k equal-sized subsets: one subset used as the test data and the remaining $k-1$ as the training data. The process is then repeated k times, with each of the subsets serving as the test data. In leave-one-out cross-validation, k-fold cross-validation is applied, where k is the total number of data instances. In Bootstrapping, the data are sampled with replacement [175], and thus the test data may contain repeated instances.

There are several widely-adopted correctness measurements such as accuracy, precision, recall, and Area Under Curve (AUC). There has been work [176] analysing the disadvantages of each measurement criterion. For example, accuracy does not distinguish between the types of errors it makes (False Positive versus False Negatives). Precision and Recall may be misled when data is unbalanced. An implication of this work is that we should carefully choose performance metrics. Chen *et al.* studied the variability of both training data and test data when assessing the correctness of an ML classifier [177]. They derived analytical expressions for the variance of the estimated performance and provided an open-source software implemented with an efficient computation algorithm. They also studied the performance of different statistical methods when comparing AUC, and found that the F-test has the best performance [178].

To better capture correctness problems, Qin *et al.* [136] proposed to generate a mirror program from the training data, and then use the behaviours this mirror program to serve as a correctness oracle. The mirror program is expected to have similar behaviours as the test data.

There has been a study of the prevalence of correctness problems among all the reported ML bugs: Zhang *et al.* [160] studied 175 Tensorflow bug reports from StackOverflow Question and Answer (QA) pages and from Github projects. Among the 175 bugs, 40 of them concern poor correctness.

Additionally, there have been many works on detecting data bug that may lead to low correctness [179], [180], [181] (see more in Section 7.1), test input or test oracle design [116], [120], [121], [123], [130], [154], [162], and test tool design [165], [167], [182] (see more in Section 5).

6.2 Model Relevance

Model relevance evaluation detects mismatches between model and data. A poor model relevance is usually associated with overfitting or underfitting. When a model is too complex for the data, even the noise of training data is fitted by the model [183]. Overfitting can easily happen, especially when the training data is insufficient, [184], [185], [186].

Cross-validation is traditionally considered to be a useful way to detect overfitting. However, it is not always clear how much overfitting is acceptable and cross-validation may be unlikely to detect overfitting if the test data is unrepresentative of potential unseen data.

Zhang *et al.* [54] introduced Perturbed Model Validation to help model selection. PMV injects noise to the training data, re-trains the model against the perturbed data, then uses the training accuracy decrease rate to detect overfitting/underfitting. The intuition is that an overfitted learner tends to fit noise in the training sample, while an underfitted learner will have low training accuracy regardless the presence of injected noise. Thus, both overfitting and underfitting tend to be less insensitive to noise and exhibit a small accuracy decrease rate against noise degree on perturbed data. PMV was evaluated on four real-world datasets (breast cancer, adult, connect-4, and MNIST) and nine synthetic datasets in the classification setting. The results reveal that PMV has better performance and provides a more

8. Sometimes a validation set is also needed to help train the model, in which circumstances the validation set will be isolated from the training set.

recognisable signal for detecting both overfitting/underfitting compared to 10-fold cross-validation.

An ML system usually gathers new data after deployment, which will be added into the training data to improve correctness. The test data, however, cannot be guaranteed to represent the future data. Werpachowski *et al.* [42] presents an overfitting detection approach via generating adversarial examples from test data. If the reweighted error estimate on adversarial examples is sufficiently different from that of the original test set, overfitting is detected. They evaluated their approach on ImageNet and CIFAR-10.

Gossmann *et al.* [187] studied the threat of test data reuse practice in the medical domain with extensive simulation studies, and found that the repeated reuse of the same test data will inadvertently result in overfitting under all considered simulation settings.

Kirk [51] mentioned that we could use training time as a complexity proxy for an ML model; it is better to choose the algorithm with equal correctness but relatively small training time.

Ma *et al.* [162] tried to relieve the overfitting problem via re-sampling the training data. Their approach was found to improve test accuracy from 75 to 93 percent on average, based on an evaluation using three image classification datasets.

6.3 Robustness and Security

6.3.1 Robustness Measurement Criteria

Unlike correctness or overfitting, robustness is a non-functional characteristic of a machine learning system. A natural way to measure robustness is to check the correctness of the system with the existence of noise [137]; a robust system should maintain performance in the presence of noise.

Moosavi-Dezfooli *et al.* [188] proposed DeepFool that computes perturbations (added noise) that 'fool' deep networks so as to quantify their robustness. Bastani *et al.* [189] presented three metrics to measure robustness: 1) pointwise robustness, indicating the minimum input change a classifier fails to be robust; 2) adversarial frequency, indicating how often changing an input changes a classifier's results; 3) adversarial severity, indicating the distance between an input and its nearest adversarial example.

Carlini and Wagner [190] created a set of attacks that can be used to construct an upper bound on the robustness of a neural network. Tjeng *et al.* [137] proposed to use the distance between a test input and its closest adversarial example to measure robustness. Ruan *et al.* [191] provided global robustness lower and upper bounds based on the test data to quantify the robustness. Gopinath *et al.* [192] proposed DeepSafe, a data-driven approach for assessing DNN robustness: inputs that are clustered into the same group should share the same label.

More recently, Mangal *et al.* [193] proposed the definition of probabilistic robustness. Their work used abstract interpretation to approximate the behaviour of a neural network and to compute an over-approximation of the input regions where the network may exhibit non-robust behaviour.

Banerjee *et al.* [194] explored the use of Bayesian Deep Learning to model the propagation of errors inside deep neural networks to mathematically model the sensitivity of neural networks to hardware errors, without performing extensive fault injection experiments.

6.3.2 Perturbation Targeting Test Data

The existence of adversarial examples allows attacks that may lead to serious consequences in safety-critical applications such as self-driving cars. There is a whole separate literature on adversarial example generation that deserves a survey of its own, and so this paper does not attempt to fully cover it. Rather, we focus on those promising aspects that could be fruitful areas for future research at the intersection of traditional software testing and machine learning.

Carlini and Wagner [190] developed adversarial example generation approaches using distance metrics to quantify similarity. The approach succeeded in generating adversarial examples for all images on the recently proposed defensively distilled networks [195].

Adversarial input generation has been widely adopted to test the robustness of autonomous driving systems [1], [76], [79], [92], [93]. There has also been research on generating adversarial inputs for NLI models [98], [99](Section 8.3), malware detection [169], and Differentiable Neural Computer [100].

Papernot *et al.* [196], [197] designed a library to standardise the implementation of adversarial example construction. They pointed out that standardising adversarial example generation is important because 'benchmarks constructed without a standardised implementation of adversarial example construction are not comparable to each other': it is not easy to tell whether a good result is caused by a high level of robustness or by the differences in the adversarial example construction procedure.

Other techniques to generate test data that check the neural network robustness include symbolic execution [108], [111], fuzz testing [90], combinatorial Testing [148], and abstract interpretation [193]. In Section 5.1, we cover these test generation techniques in more detail.

6.3.3 Perturbation Targeting the Whole System

Jha *et al.* [198] presented AVFI, which used application/software fault injection to approximate hardware errors in the sensors, processors, or memory of the autonomous vehicle (AV) systems to test the robustness. They also presented Kayotee [199], a fault injection-based tool to systematically inject faults into software and hardware components of the autonomous driving systems. Compared with AVFI, Kayotee is capable of characterising error propagation and masking using a closed-loop simulation environment, which is also capable of injecting bit flips directly into GPU and CPU architectural state. DriveFI [96], further presented by Jha *et al.*, is a fault-injection engine that mines situations and faults that maximally impact AV safety.

Tuncali *et al.* [102] considered the closed-loop behaviour of the whole system to support adversarial example generation for autonomous driving systems, not only in image space, but also in configuration space.

6.4 Efficiency

The empirical study of Zhang *et al.* [160] on Tensorflow bug-related artefacts (from StackOverflow QA page and Github) found that nine out of 175 ML bugs (5.1 percent) belong to efficiency problems. The reasons may either be that efficiency problems rarely occur or that these issues are difficult to detect.

Kirk [51] pointed out that it is possible to use the efficiency of different machine learning algorithms when training the model to compare their complexity.

Spieker and Gotlieb [200] studied three training data reduction approaches, the goal of which was to find a smaller subset of the original training data set with similar characteristics during model training, so that model building speed could be improved for faster machine learning testing.

6.5 Fairness

Fairness is a relatively recently-emerging non-functional characteristic. According to the work of Barocas and Selbst [201], there are the following five major causes of unfairness.

1) *Skewed sample*: once some initial bias happens, such bias may compound over time.
2) *Tainted examples*: the data labels are biased because of biased labelling activities of humans.
3) *Limited features*: features may be less informative or reliably collected, misleading the model in building the connection between the features and the labels.
4) *Sample size disparity*: if the data from the minority group and the majority group are highly imbalanced, ML model may the minority group less well.
5) *Proxies*: some features are proxies of sensitive attributes (e.g., neighbourhood in which a person resides), and may cause bias to the ML model even if sensitive attributes are excluded.

Research on fairness focuses on measuring, discovering, understanding, and coping with the observed differences regarding different groups or individuals in performance. Such differences are associated with fairness bugs, which can offend and even harm users, and cause programmers and businesses embarrassment, mistrust, loss of revenue, and even legal violations [171].

6.5.1 Fairness Definitions and Measurement Metrics

There are several definitions of fairness proposed in the literature, yet no firm consensus [202], [203], [204], [205]. Nevertheless, these definitions can be used as oracles to detect fairness violations in ML testing.

To help illustrate the formalisation of ML fairness, we use X to denote a set of individuals, Y to denote the true label set when making decisions regarding each individual in X. Let h be the trained machine learning predictive model. Let A be the set of sensitive attributes, and Z be the remaining attributes.

1) Fairness Through Unawareness. Fairness Through Unawareness (FTU) means that an algorithm is fair so long as the protected attributes are not explicitly used in the decision-making process [206]. It is a relatively low-cost way to define and ensure fairness. Nevertheless, sometimes the non-sensitive attributes in X may contain information correlated to sensitive attributes that may thereby lead to discrimination [202], [206]. Excluding sensitive attributes may also impact model accuracy and yield less effective predictive results [207].

2) Group Fairness. A model under test has group fairness if groups selected based on sensitive attributes have an equal probability of decision outcomes. There are several types of group fairness.

Demographic Parity is a popular group fairness measurement [208]. It is also named *Statistical Parity* or *Independence Parity*. It requires that a decision should be independent of the protected attributes. Let G_1 and G_2 be the two groups belonging to X divided by a sensitive attribute $a \in A$. A model h under test satisfies demographic parity if $P\{h(x_i) = 1|x_i \in G_1\} = P\{h(x_j) = 1|x_j \in G_2\}$.

Equalised Odds is another group fairness approach proposed by Hardt *et al.* [139]. A model under test h satisfies *Equalised Odds* if h is independent of the protected attributes when a target label Y is fixed as y_i: $P\{h(x_i) = 1|x_i \in G_1, Y = y_i\} = P\{h(x_j) = 1|x_j \in G_2, Y = y_i\}$.

When the target label is set to be positive, Equalised Odds becomes *Equal Opportunity* [139]. It requires that the true positive rate should be the same for all the groups. A model h satisfies *Equal Opportunity* if h is independent of the protected attributes when a target class Y is fixed as being positive: $P\{h(x_i) = 1|x_i \in G_1, Y = 1\} = P\{h(x_j) = 1|x_j \in G_2, Y = 1\}$.

3) Counter-Factual Fairness. Kusner *et al.* [206] introduced *Counter-factual Fairness*. A model satisfies Counter-factual Fairness if its output remains the same when the protected attribute is flipped to a counter-factual value, and other variables are modified as determined by the assumed causal model. Let a be a protected attribute, a' be the counterfactual attribute of a, x_i' be the new input with a changed into a'. Model h is counter-factually fair if, for any input x_i and protected attribute a:$P\{h(x_i)_a = y_i|a \in A, x_i \in X\} = P\{h(x_i')_{a'} = y_i|a \in A, x_i \in X\}$. This measurement of fairness additionally provides a mechanism to interpret the causes of bias, because the variables other than the protected attributes are controlled, and thus the differences in $h(x_i)$ and $h(x_i')$ must be caused by variations in A.

4) Individual Fairness. Dwork *et al.* [138] proposed a use task-specific similarity metric to describe the pairs of individuals that should be regarded as similar. According to Dwork *et al.*, a model h with individual fairness should give similar predictive results among similar individuals: $P\{h(x_i)|x_i \in X\} = P\{h(x_j) = y_i|x_j \in X\} \quad \text{iff} \quad d(x_i, x_j) < \epsilon$, where d is a distance metric for individuals that measures their similarity, and ϵ is tolerance to such differences.

Analysis and Comparison of Fairness Metrics. Although there are many existing definitions of fairness, each has its advantages and disadvantages. Which fairness is the most suitable remains controversial. There is thus some work surveying and analysing the existing fairness metrics, or investigate and compare their performance based on experimental results, as introduced below.

Gajane and Pechenizkiy [202] surveyed how fairness is defined and formalised in the literature. Corbett-Davies and Goel [62] studied three types of fairness definitions: anti-classification, classification parity, and calibration. They pointed out the deep statistical limitations of each type with examples. Verma and Rubin [203] explained and illustrated the existing most prominent fairness definitions based on a common, unifying dataset.

Saxena *et al.* [204] investigated people's perceptions of three of the fairness definitions. About 200 recruited participants from Amazon's Mechanical Turk were asked to choose their preference over three allocation rules on two individuals having each applied for a loan. The results demonstrate a

clear preference for the way of allocating resources in proportion to the applicants' loan repayment rates.

Support for Fairness Improvement. Metevier *et al.* [209] proposed RobinHood, an algorithm that supports multiple user-defined fairness definitions under the scenario of offline contextual bandits.[9] RobinHood makes use of concentration inequalities [211] to calculate high-probability bounds and to search for solutions that satisfy the fairness requirements. It gives user warnings when the requirements are violated. The approach is evaluated under three application scenarios: a tutoring system, a loan approval setting, and the criminal recidivism, all of which demonstrate the superiority of RobinHood over other algorithms.

Albarghouthi and Vinitsky [75] proposed the concept of 'fairness-aware programming', in which fairness is a first-class concern. To help developers define their own fairness specifications, they developed a specification language. Like assertions in traditional testing, the fairness specifications are developed into the run-time monitoring code to enable multiple executions to catch violations. A prototype was implemented in Python.

Agarwal *et al* . [212] proposed to reduce fairness classification into a problem of cost-sensitive classification (where the costs of different types of errors are differentiated). The application scenario is binary classification, with the underlying classification method being treated as a black box. The reductions optimise the trade-off between accuracy and fairness constraints.

Albarghouthi *et al* . [168] proposed an approach to repair decision-making programs using distribution-guided inductive synthesis.

6.5.2 Test Generation Techniques for Fairness Testing

Galhotra *et al.* [5], [213] proposed Themis which considers group fairness using causal analysis [214]. It defines fairness scores as measurement criteria for fairness and uses random test generation techniques to evaluate the degree of discrimination (based on fairness scores). Themis was also reported to be more efficient on systems that exhibit more discrimination.

Themis generates tests randomly for group fairness, while Udeshi *et al.* [101] proposed Aequitas, focusing on test generation to uncover discriminatory inputs and those inputs essential to understand individual fairness. The generation approach first randomly samples the input space to discover the presence of discriminatory inputs, then searches the neighbourhood of these inputs to find more inputs. As well as detecting fairness bugs, *Aeqitas* also retrains the machine-learning models and reduce discrimination in the decisions made by these models.

Agarwal *et al.* [109] used symbolic execution together with local explainability to generate test inputs. The key idea is to use the local explanation, specifically Local Interpretable Model-agnostic Explanations[10] to identify whether factors that drive decisions include protected attributes. The evaluation indicates that the approach generates 3.72 times more successful test cases than THEMIS across 12 benchmarks.

Tramer *et al.* [171] were the first to proposed the concept of 'fairness bugs'. They consider a statistically significant association between a protected attribute and an algorithmic output to be a fairness bug, specially named 'Unwarranted Associations' in their paper. They proposed the first comprehensive testing tool, aiming to help developers test and debug fairness bugs with an 'easily interpretable' bug report. The tool is available for various application areas including image classification, income prediction, and health care prediction.

Sharma and Wehrheim [122] sought to identify causes of unfairness via checking whether the algorithm under test is sensitive to training data changes. They mutated the training data in various ways to generate new datasets, such as changing the order of rows, columns, and shuffling feature names and values. 12 out of 14 classifiers were found to be sensitive to these changes.

6.6 Interpretability

Manual Assessment of Interpretability. The existing work on empirically assessing the interpretability property usually includes humans in the loop. That is, manual assessment is currently the primary approach to evaluate interpretability. Doshi-Velez and Kim [65] gave a taxonomy of evaluation (testing) approaches for interpretability: application-grounded, human-grounded, and functionally-grounded. Application-grounded evaluation involves human experimentation with a real application scenario. Human-grounded evaluation uses results from human evaluation on simplified tasks. Functionally-grounded evaluation requires no human experiments but uses a quantitative metric as a proxy for explanation quality, for example, a proxy for the explanation of a decision tree model may be the depth of the tree.

Friedler *et al.* [215] introduced two types of interpretability: global interpretability means understanding the entirety of a trained model; local interpretability means understanding the results of a trained model on a specific input and the corresponding output. They asked 1,000 users to produce the expected output changes of a model given an input change, and then recorded accuracy and completion time over varied models. Decision trees and logistic regression models were found to be more locally interpretable than neural networks.

Automatic Assessment of Interpretability. Cheng *et al.* [46] presented a metric to understand the behaviours of an ML model. The metric measures whether the learner has learned the object in object identification scenario via occluding the surroundings of the objects.

Christoph [70] proposed to measure interpretability based on the category of ML algorithms. He claimed that 'the easiest way to achieve interpretability is to use only a subset of algorithms that create interpretable models'. He identified several models with good interpretability, including linear regression, logistic regression and decision tree models.

Zhou *et al.* [216] defined the concepts of Metamorphic Relation Patterns (MRPs) and Metamorphic Relation Input Patterns (MRIPs) that can be adopted to help end users

9. A contextual bandit is a type of algorithm that learns to take actions based on rewards such as user click rate [210].

10. Local Interpretable Model-agnostic Explanations produces decision trees corresponding to an input that could provide paths in symbolic execution [110]

understand how an ML system works. They conducted case studies of various systems, including large commercial websites, Google Maps navigation, Google Maps location-based search, image analysis for face recognition (including Facebook, MATLAB, and OpenCV), and the Google video analysis service Cloud Video Intelligence.

Evaluation of Interpretability Improvement Methods. Machine learning classifiers are widely used in many medical applications, yet the clinical meaning of the predictive outcome is often unclear. Chen *et al.* [217] investigated several interpretability-improving methods which transform classifier scores to a probability of disease scale. They showed that classifier scores on arbitrary scales can be calibrated to the probability scale without affecting their discrimination performance.

6.7 Privacy

Ding *et al* . [218] treat programs as grey boxes, and detect differential privacy violations via statistical tests. For the detected violations, they generate counter examples to illustrate these violations as well as to help developers understand and fix bugs. Bichsel *et al* . [219] proposed to estimate the ϵ parameter in differential privacy, aiming to find a triple (x, x', Φ) that witnesses the largest possible privacy violation, where x and x' are two test inputs and Φ is a possible set of outputs.

7 ML Testing Components

This section organises the work on ML testing by identifying the component (data, learning program, or framework) for which ML testing may reveal a bug.

7.1 Bug Detection in Data

Data is a 'component' to be tested in ML testing, since the performance of the ML system largely depends on the data. Furthermore, as pointed out by Breck *et al.* [45], it is important to detect data bugs early because predictions from the trained model are often logged and used to generate further data. This subsequent generation creates a feedback loop that may amplify even small data bugs over time.

Nevertheless, data testing is challenging [220]. According to the study of Amershi *et al.* [8], the management and evaluation of data is among the most challenging tasks when developing an AI application in Microsoft. Breck *et al.* [45] mentioned that data generation logic often lacks visibility in the ML pipeline; the data are often stored in a raw-value format (e.g., CSV) that strips out semantic information that can help identify bugs.

7.1.1 Bug Detection in Training Data

Rule-Based Data Bug Detection. Hynes *et al.* [179] proposed *data linter*– an ML tool, inspired by code linters, to automatically inspect ML datasets. They considered three types of data problems: 1) miscoded data, such as mistyping a number or date as a string; 2) outliers and scaling, such as uncommon list length; 3) packaging errors, such as duplicate values, empty examples, and other data organisation issues.

Cheng *et al.* [46] presented a series of metrics to evaluate whether the training data have covered all important scenarios.

Performance-Based Data Bug Detection. To solve the problems in training data, Ma *et al.* [162] proposed MODE. MODE identifies the 'faulty neurons' in neural networks that are responsible for the classification errors, and tests the training data via data resampling to analyse whether the faulty neurons are influenced. MODE allows to improve test effectiveness from 75 to 93 percent on average, based on evaluation using the MNIST, Fashion MNIST, and CIFAR-10 datasets.

7.1.2 Bug Detection in Test Data

Metzen *et al.* [221] proposed to augment DNNs with a small sub-network, specially designed to distinguish genuine data from data containing adversarial perturbations. Wang *et al.* [222] used DNN model mutation to expose adversarial examples motivated by their observation that adversarial samples are more sensitive to perturbations [223]. The evaluation was based on the MNIST and CIFAR10 datasets. The approach detects 96.4/90.6 percent adversarial samples with 74.1/86.1 mutations for MNIST/CIFAR10.

Adversarial examples in test data raise security risks. Detecting adversarial examples is thereby similar to bug detection. Carlini and Wagner [224] surveyed ten proposals that are designed for detecting adversarial examples and compared their efficacy. They found that detection approaches rely on loss functions and can thus be bypassed when constructing new loss functions. They concluded that adversarial examples are significantly harder to detect than previously appreciated.

The insufficiency of the test data may not able to detect overfitting issues, and could also be regarded as a type of data bugs. The approaches for detecting test data insufficiency were discussed in Section 5.3, such as coverage [1], [76], [92] and mutation score [152].

7.1.3 Skew Detection in Training and Test Data

The training instances and the instances that the model predicts should exhibit consistent features and distributions. Kim *et al.* [127] proposed two measurements to evaluate the skew between training and test data: one is based on Kernel Density Estimation to approximate the likelihood of the system having seen a similar input during training, the other is based on the distance between vectors representing the neuron activation traces of the given input and the training data (e.g., euclidean distance).

Breck [45] investigated the skew in training data and serving data (the data that the ML model predicts after deployment). To detect the skew in features, they do key-join feature comparison. To quantify the skew in distribution, they argued that general approaches such as KL divergence or cosine similarity might not be sufficiently intuitive for produce teams. Instead, they proposed to use the largest change in probability as a value in the two distributions as a measurement of their distance.

7.1.4 Frameworks in Detecting Data Bugs

Breck *et al.* [45] proposed a data validation system for detecting data bugs. The system applies constraints (e.g., type, domain, valency) to find bugs in single-batch (within the

training data or new data), and quantifies the distance between training data and new data. Their system is deployed as an integral part of TFX (an end-to-end machine learning platform at Google). The deployment in production provides evidence that the system helps early detection and debugging of data bugs. They also summarised the type of data bugs, in which new feature column, unexpected string values, and missing feature columns are the three most common.

Krishnan *et al.* [225], [226] proposed a model training framework, ActiveClean, that allows for iterative data cleaning while preserving provable convergence properties. ActiveClean suggests a sample of data to clean based on the data's value to the model and the likelihood that it is 'dirty'. The analyst can then apply value transformations and filtering operations to the sample to 'clean' the identified dirty data.

In 2017, Krishnan *et al.* [180] presented a system named BoostClean to detect domain value violations (i.e., when an attribute value is outside of an allowed domain) in training data. The tool utilises the available cleaning resources such as Isolation Forest [227] to improve a model's performance. After resolving the problems detected, the tool is able to improve prediction accuracy by up to 9 percent in comparison to the best non-ensemble alternative.

ActiveClean and BoostClean may involve a human in the loop of testing process. Schelter *et al.* [181] focus on the automatic 'unit' testing of large-scale datasets. Their system provides a declarative API that combines common as well as user-defined quality constraints for data testing. Krishnan and Wu [228] also targeted automatic data cleaning and proposed AlphaClean. They used a greedy tree search algorithm to automatically tune the parameters for data cleaning pipelines. With AlphaClean, the user could focus on defining cleaning requirements and let the system find the best configuration under the defined requirement. The evaluation was conducted on three datasets, demonstrating that AlphaClean finds solutions of up to 9X better than state-of-the-art parameter tuning methods.

Training data testing is also regarded as a part of a whole machine learning workflow in the work of Baylor *et al.* [182]. They developed a machine learning platform that enables data testing, based on a property description schema that captures properties such as the features present in the data and the expected type or valency of each feature.

There are also data cleaning technologies such as statistical or learning approaches from the domain of traditional database and data warehousing. These approaches are not specially designed or evaluated for ML, but they can be repurposed for ML testing [229].

7.2 Bug Detection in Learning Program

Bug detection in the learning program checks whether the algorithm is correctly implemented and configured, e.g., the model architecture is designed well, and whether there exist coding errors.

Unit Tests for ML Learning Program. McClure [230] introduced ML unit testing with TensorFlow built-in testing functions to help ensure that 'code will function as expected' to help build developers' confidence.

Schaul *et al.* [231] developed a collection of unit tests specially designed for stochastic optimisation. The tests are small-scale, isolated with well-understood difficulty. They could be adopted in the beginning learning stage to test the learning algorithms to detect bugs as early as possible.

Algorithm Configuration Examination. Sun *et al.* [232] and Guo *et al.* [233] identified operating systems, language, and hardware Compatibility issues. Sun *et al.* [232] studied 329 real bugs from three machine learning frameworks: Scikit-learn, Paddle, and Caffe. Over 22 percent bugs are found to be compatibility problems due to incompatible operating systems, language versions, or conflicts with hardware. Guo *et al.* [233] investigated deep learning frameworks such as TensorFlow, Theano, and Torch. They compared the learning accuracy, model size, robustness with different models classifying dataset MNIST and CIFAR-10.

The study of Zhang *et al.* [160] indicates that the most common learning program bug is due to the change of TensorFlow API when the implementation has not been updated accordingly. Additionally, 23.9 percent (38 in 159) of the bugs from ML projects in their study built based on TensorFlow arise from problems in the learning program.

Karpov *et al.* [234] also highlighted testing algorithm parameters in all neural network testing problems. The parameters include the number of neurons and their types based on the neuron layer types, the ways the neurons interact with each other, the synapse weights, and the activation functions. However, the work currently remains unevaluated.

Algorithm Selection Examination. Developers usually have more than one learning algorithm to choose from. Fu and Menzies [235] compared deep learning and classic learning on the task of linking Stack Overflow questions, and found that classic learning algorithms (such as refined SVM) could achieve similar (and sometimes better) results at a lower cost. Similarly, the work of Liu *et al.* [236] found that the k-Nearest Neighbours (KNN) algorithm achieves similar results to deep learning for the task of commit message generation.

Mutant Simulations of Learning Program Faults. Murphy *et al.* [117], [128] used mutants to simulate programming code errors to investigate whether the proposed metamorphic relations are effective at detecting errors. They introduced three types of mutation operators: switching comparison operators, mathematical operators, and off-by-one errors for loop variables.

Dolby *et al.* [237] extended WALA to support static analysis of the behaviour of tensors in Tensorflow learning programs written in Python. They defined and tracked tensor types for machine learning, and changed WALA to produce a dataflow graph to abstract possible program behavours.

7.3 Bug Detection in Framework

The current research on framework testing focuses on studying framework bugs (Section 7.3.1) and detecting bugs in framework implementation (Section 7.3.2).

7.3.1 Study of Framework Bugs

Xiao *et al.* [238] focused on the security vulnerabilities of popular deep learning frameworks including Caffe, TensorFlow, and Torch. They examined the code of popular deep learning frameworks. The dependency of these frameworks was found to be very complex. Multiple vulnerabilities were identified in their implementations. The most common

vulnerabilities are bugs that cause programs to crash, non-terminate, or exhaust memory.

Guo *et al.* [233] tested deep learning frameworks, including TensorFlow, Theano, and Torch, by comparing their runtime behaviour, training accuracy, and robustness, under identical algorithm design and configuration. The results indicate that runtime training behaviours are different for each framework, while the prediction accuracies remain similar.

Low Efficiency is a problem for ML frameworks, which may directly lead to inefficiency of the models built on them. Sun *et al.* [232] found that approximately 10 percent of reported framework bugs concern low efficiency. These bugs are usually reported by users. Compared with other types of bugs, they may take longer for developers to resolve.

7.3.2 Implementation Testing of Frameworks

Many learning algorithms are implemented inside ML frameworks. Implementation bugs in ML frameworks may cause neither crashes, errors, nor efficiency problems [239], making their detection challenging.

Challenges in Implementation Bug Detection. Thung *et al.* [159] studied machine learning bugs in 2012. Their results, regarding 500 bug reports from three machine learning systems, indicated that approximately 22.6 percent bugs are due to incorrect algorithm implementations. Cheng *et al.* [240] injected implementation bugs into classic machine learning code in Weka and observed the performance changes that resulted. They found that 8 to 40 percent of the logically non-equivalent executable mutants (injected implementation bugs) were statistically indistinguishable from their original versions.

Solutions for Implementation Bug Detection. Some work has used multiple implementations or differential testing to detect bugs. For example, Alebiosu *et al.* [135] found five faults in 10 Naive Bayes implementations and four faults in 20 k-nearest neighbour implementations. Pham *et al.* [48] found 12 bugs in three libraries (i.e., TensorFlow, CNTK, and Theano), 11 datasets (including ImageNet, MNIST, and KGS Go game), and 30 pre-trained models (see Section 5.2.2).

However, not every algorithm has multiple implementations. Murphy *et al.* [10], [115] were the first to discuss the possibilities of applying metamorphic relations to machine learning implementations. They listed several transformations of the input data that should ought not to bring changes in outputs, such as multiplying numerical values by a constant, permuting or reversing the order of the input data, and adding additional data. Their case studies found that their metamorphic relations held on three machine learning applications.

Xie *et al.* [118] focused on supervised learning. They proposed to use more specific metamorphic relations to test the implementations of supervised classifiers. They discussed five types of potential metamorphic relations on KNN and Naive Bayes on randomly generated data. In 2011, they further evaluated their approach using mutated machine learning code [241]. Among the 43 injected faults in Weka [119] (injected by MuJava [242]), the metamorphic relations were able to reveal 39. In their work, the test inputs were randomly generated data.

Dwarakanath *et al.* [121] applied metamorphic relations to find implementation bugs in image classification. For classic machine learning such as SVM, they conducted mutations such as changing feature or instance orders, and linear scaling of the test features. For deep learning models such as residual networks (which the data features are not directly available), they proposed to normalise or scale the test data, or to change the convolution operation order of the data. These changes were intended to bring no change to the model performance when there are no implementation bugs. Otherwise, implementation bugs are exposed. To evaluate, they used MutPy to inject mutants that simulate implementation bugs, of which the proposed metamorphic relations are able to find 71 percent.

7.3.3 Study of Frameworks Test Oracles

Nejadgholi and Yang [133] studied the approximated oracles of four popular deep learning libraries: Tensorflow, Theano, PyTorch, and Keras. 5 to 24 percent oracles were found to be approximated oracles with a flexible threshold (in contrast to certain oracles). 5-27 percent of the approximated oracles used the outputs from other libraries/frameworks. Developers were also found to modify approximated oracles frequently due to code evolution.

8 Application Scenarios

Machine learning has been widely adopted in different areas. This section introduces such domain-specific testing approaches in three typical application domains: autonomous driving, machine translation, and neural language inference.

8.1 Autonomous Driving

Testing autonomous vehicles has a comparatively long history. For example, in 2004, Wegener and Bühler compared different fitness functions when evaluating the tests of autonomous car parking systems [243]. Testing autonomous vehicles also has many research opportunities and open questions, as pointed out and discussed by Woehrle *et al.* [244].

More recently, search-based test generation for AV testing has been successfully applied. Abdessalem *et al.* [245], [246] focused on improving the efficiency and accuracy of search-based testing of advanced driver assistance systems (ADAS) in AVs. Their algorithms use classification models to improve the efficiency of the search-based test generation for critical scenarios. Search algorithms are further used to refine classification models to improve their accuracy. Abdessalem *et al.* [247] also proposed FITEST, a multi-objective search algorithm that searches feature interactions which violate system requirements or lead to failures.

Most of the current autonomous vehicle systems that have been put into the market are semi-autonomous vehicles, which require a human driver to serve as a fall-back [161], as was the case with the work of Wegener and Bühler [243]. An issue that causes the human driver to take control of the vehicle is called a *disengagement*.

Banerjee *et al.* [161] investigated the causes and impacts of 5,328 disengagements from the data of 12 AV manufacturers for 144 vehicles that drove a cumulative 1,116,605 autonomous miles, 42 (0.8 percent) of which led to accidents. They classified the causes of disengagements into 10 types. 64 percent of the disengagements were found to be

caused by the bugs in the machine learning system, among which the behaviours of image classification (e.g., improper detection of traffic lights, lane markings, holes, and bumps) were the dominant causes accounting for 44 percent of all reported disengagements. The remaining 20 percent were due to the bugs in the control and decision framework such as improper motion planning.

Pei *et al.* [1] used gradient-based differential testing to generate test inputs to detect potential DNN bugs and leveraged neuron coverage as a guideline. Tian *et al.* [76] proposed to use a set of image transformation to generate tests, which simulate the potential noise that could be present in images obtained from a real-world camera. Zhang *et al.* [79] proposed DeepRoad, a GAN-based approach to generate test images for real-world driving scenes. Their approach is able to support two weather conditions (i.e., snowy and rainy). The images were generated with the pictures from YouTube videos. Zhou *et al.* [81] proposed DeepBillboard, which generates real-world adversarial billboards that can trigger potential steering errors of autonomous driving systems. It demonstrates the possibility of generating continuous and realistic physical-world tests for practical autonomous-driving systems.

Wicker *et al.* [93] used feature-guided Monte Carlo Tree Search to identify elements of an image that are most vulnerable to a self-driving system; adversarial examples. Jha *et al.* [96] accelerated the process of finding 'safety-critical' issues via analytically modelling the injection of faults into an AV system as a Bayesian network. The approach trains the network to identify safety critical faults automatically. The evaluation was based on two production-grade AV systems from NVIDIA and Baidu, indicating that the approach can find many situations where faults lead to safety violations.

Uesato *et al.* [94] aimed to find catastrophic failures in safety-critical agents like autonomous driving in reinforcement learning. They demonstrated the limitations of traditional random testing, then proposed a predictive adversarial example generation approach to predict failures and estimate reliable risks. The evaluation on TORCS simulator indicates that the proposed approach is both effective and efficient with fewer Monte Carlo runs.

To test whether an algorithm can lead to a problematic model, Dreossi *et al.* [170] proposed to generate training data as well as test data. Focusing on Convolutional Neural Networks (CNN), they build a tool to generate natural images and visualise the gathered information to detect blind spots or corner cases under the autonomous driving scenario. Although there is currently no evaluation, the tool has been made available.[11]

Tuncali *et al.* [102] presented a framework that supports both system-level testing and the testing of those properties of an ML component. The framework also supports fuzz test input generation and search-based testing using approaches such as Simulated Annealing and Cross-Entropy optimisation.

While many other studies investigated DNN model testing for research purposes, Zhou *et al.* [95] combined fuzzing and metamorphic testing to test LiDAR, which is an obstacle-perception module of real-life self-driving cars, and detected real-life fatal bugs.

Jha *et al.* presented AVFI [198] and Kayotee [199], which are fault injection-based tools to systematically inject faults into autonomous driving systems to assess their safety and reliability.

O'Kelly *et al* . [72] proposed a 'risk-based framework' for AV testing to predict the probability of an accident in a base distribution of traffic behaviour (derived from the public traffic data collected by the US Department of Transportation). They argued that formally verifying correctness of an AV system is infeasible due to the challenge of formally defining "correctness" as well as the white-box requirement. Traditional testing AVs in a real environment requires prohibitive amounts of time. To tackle these problems, they view AV testing as rare-event simulation problem, then evaluate the accident probability to accelerate AV testing.

8.2 Machine Translation

Machine translation automatically translates text or speech from one language to another. The BiLingual Evaluation Understudy (BLEU) score [248] is a widely-adopted measurement criterion to evaluate machine translation quality. It assesses the correspondence between a machine's output and that of a human.

Zhou *et al.* [129], [130] used self-defined metamorphic relations in their tool 'MT4MT' to test the translation consistency of machine translation systems. The idea is that some changes to the input should not affect the overall structure of the translated output. Their evaluation showed that Google Translate outperformed Microsoft Translator for long sentences whereas the latter outperformed the former for short and simple sentences. They hence suggested that the quality assessment of machine translations should consider multiple dimensions and multiple types of inputs.

Sun *et al.* [86] combine mutation testing and metamorphic testing to test and repair the consistency of machine translation systems. Their approach, TransRepair, enables automatic test input generation, automatic test oracle generation, as well as automatic translation repair. They first applied mutation on sentence inputs to find translation inconsistency bugs, then used translations of the mutated sentences to optimise the translation results in a black-box or grey-box manner. Evaluation demonstrates that TransRepair fixes 28 and 19 percent bugs on average for Google Translate and Transformer.

Compared with existing model retraining approaches, TransRepair has the following advantages: 1) more effective than data augmentation; 2) source code in dependant (black box); 3) computationally cheap (avoids space and time expense of data collection and model retraining); 4) flexible (enables repair without touching other well-formed translations).

The work of Zheng *et al.* [249], [250], [251] proposed two algorithms for detecting two specific types of machine translation violations: (1) under-translation, where some words/phrases from the original text are missing in the translation, and (2) over-translation, where some words/phrases from the original text are unnecessarily translated multiple times. The algorithms are based on a statistical analysis of both the original texts and the translations, to check whether there are violations of one-to-one mappings in words/phrases.

11. https://github.com/shromonag/FalsifyNN

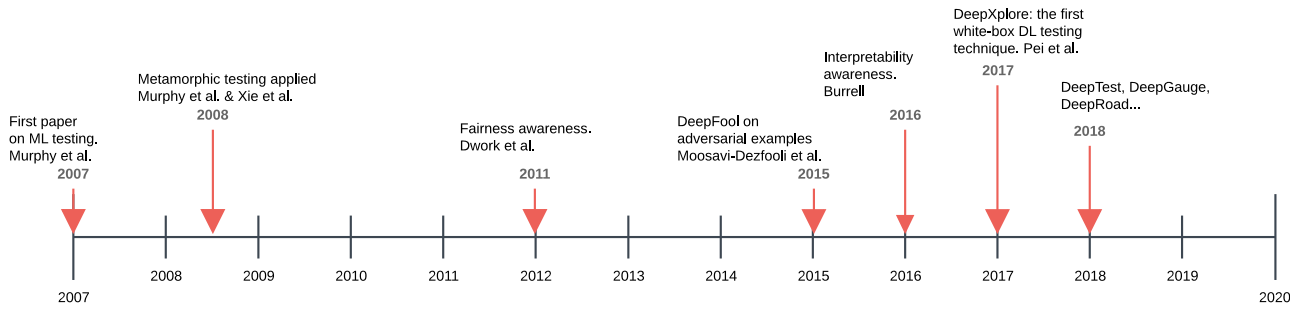


Fig. 8. Timeline of ML testing research.

8.3 Natural Language Inference

A Nature Language Inference task judges the inference relationship of a pair of natural language sentences. For example, the sentence 'A person is in the room' could be inferred from the sentence 'A girl is in the room'.

Several works have tested the robustness of NLI models. Nie *et al.* [98] generated sentence mutants (called 'rule-based adversaries' in the paper) to test whether the existing NLI models have semantic understanding. Seven state-of-the-art NLI models (with diverse architectures) were all unable to recognise simple semantic differences when the word-level information remains unchanged.

Similarly, Wang *et al.* [99] mutated the inference target pair by simply swapping them. The heuristic is that a good NLI model should report comparable accuracy between the original and swapped test set for contradictory pairs and for neutral pairs, but lower accuracy in swapped test set for entailment pairs (the hypothesis may or may not be true given a premise).

9 Analysis of Literature Review

This section analyses the research distribution among different testing properties and machine learning categories. It also summarises the datasets (name, description, size, and usage scenario of each dataset) that have been used in ML testing.

9.1 Timeline

Fig. 8 shows several key contributions in the development of ML testing. As early as in 2007, Murphy *et al.* [10] mentioned the idea of testing machine learning applications. They classified machine learning applications as 'non-testable' programs considering the difficulty of getting test oracles. They primarily consider the detection of implementation bugs, described as "to ensure that an application using the algorithm correctly implements the specification and fulfils the users' expectations". Afterwards, Murphy *et al.* [115] discussed the properties of machine learning algorithms that may be adopted as metamorphic relations to detect implementation bugs.

In 2009, Xie *et al.* [118] also applied metamorphic testing on supervised learning applications.

Fairness testing was proposed in 2012 by Dwork *et al.* [138]; the problem of interpretability was proposed in 2016 by Burrell [252].

In 2017, Pei *et al.* [1] published the first white-box testing paper on deep learning systems. Their work pioneered to propose coverage criteria for DNN. Enlightened by this paper, a number of machine learning testing techniques have emerged, such as DeepTest [76], DeepGauge [92], DeepConcolic [111], and DeepRoad [79]. A number of software testing techniques has been applied to ML testing, such as different testing coverage criteria [76], [92], [145], mutation testing [152], combinatorial testing [149], metamorphic testing [100], and fuzz testing [89].

9.2 Research Distribution Among Machine Learning Categories

This section introduces and compares the research status of each machine learning category.

9.2.1 Research Distribution Between General Machine Learning and Deep Learning

To explore research trends in ML testing, we classify the collected papers into two categories: those targeting only deep learning and those designed for general machine learning (including deep learning).

Among all 144 papers, 56 papers (38.9 percent) present testing techniques that are specially designed for deep learning alone; the remaining 88 papers cater for general machine learning.

We further investigated the number of papers in each category for each year, to observe whether there is a trend of moving from testing general machine learning to deep learning. Fig. 9 shows the results. Before 2017, papers mostly focus on general machine learning; after 2018, both general machine learning and deep learning specific testing notably arise.

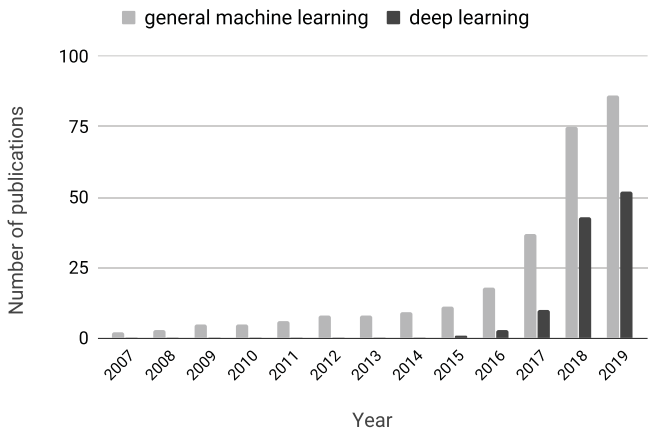


Fig. 9. Commutative trends in general machine learning and deep learning.

TABLE 4
Search Hits and Testing Distribution of Supervised/Unsupervised/Reinforcement Learning

Category	Scholar hits	Google hits	Testing hits
Supervised	1,610k	73,500k	119/144
Unsupervised	619k	17,700k	3/144
Reinforcement	2,560k	74,800k	1/144

9.2.2 Research Distribution Among Supervised/ Unsupervised/Reinforcement Learning Testing

We further classified the papers based on the three machine learning categories: 1) supervised learning testing, 2) unsupervised learning testing, and 3) reinforcement learning testing. One striking finding is that almost all the work we identified in this survey focused on testing supervised machine learning. Among the 144 related papers, there are currently only three papers testing unsupervised machine learning: Murphy *et al.* [115] introduced metamorphic relations that work for both supervised and unsupervised learning algorithms. Ramanathan and Pullum [7] proposed a combination of symbolic and statistical approaches to test the k-means clustering algorithm. Xie *et al.* [125] designed metamorphic relations for unsupervised learning. We were able to find out only one paper that focused on reinforcement learning testing: Uesato *et al.* [94] proposed a predictive adversarial example generation approach to predict failures and estimate reliable risks in reinforcement learning.

Because of this notable imbalance, we sought to understand whether there was also an imbalance of research popularity in the machine learning areas. To approximate the research popularity of each category, we searched terms 'supervised learning', 'unsupervised learning', and 'reinforcement learning' in Google Scholar and Google. Table 4 shows the results of search hits. The last column shows the number/ratio of papers that touch each machine learning category in ML testing. For example, 119 out of 144 papers were observed for supervised learning testing purpose. The table suggests that testing popularity of different categories is not related to their overall research popularity. In particular, reinforcement learning has higher search hits than supervised learning, but we did not observe any related work that conducts direct reinforcement learning testing.

There may be several reasons for this observation. First, supervised learning is a widely-known learning scenario associated with classification, regression, and ranking problems [28]. It is natural that researchers would emphasise the testing of widely-applied, known and familiar techniques at the beginning. Second, supervised learning usually has labels in the dataset. It is thereby easier to judge and analyse test effectiveness.

Nevertheless, many opportunities clearly remain for research in the widely-studied areas of unsupervised learning and reinforcement learning (we discuss more in Section 10).

9.2.3 Different Learning Tasks

ML involves different tasks such as classification, regression, clustering, and dimension reduction (see more in Section 2). The research focus on different tasks also appears to exhibit imbalance, with a large number of papers focusing on classification.

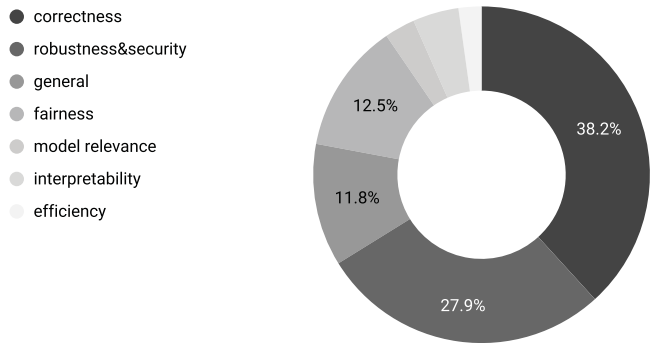


Fig. 10. Research distribution among different testing properties.

9.3 Research Distribution Among Different Testing Properties

We counted the number of papers concerning each ML testing property. Fig. 10 shows the results. The properties in the legend are ranked based on the number of papers that are specially focused on testing the associated property ('general' refers to those papers discussing or surveying ML testing generally).

From the figure, 38.7 percent of the papers test correctness. 26.8 percent of the papers focus on robustness and security problems. Fairness testing ranks the third among all the properties, with 12.0 percent of the papers.

Nevertheless, for model relevance, interpretability testing, efficiency testing, and privacy testing, fewer than 6 papers exist for each category in our paper collection.

9.4 Datasets Used in ML Testing

Tables 5, 6, 7, and 8 show details concerning widely-adopted datasets used in ML testing research. In each table, the first column shows the name and link of each dataset. The next three columns give a brief description, the size (the "+" connects training data and test data if applicable), the testing problem(s), the usage application scenario of each dataset.[12]

Table 5 shows the datasets used for image classification tasks. Datasets can be large (e.g., more than 1.4 million images in ImageNet). The last six rows show the datasets collected for autonomous driving system testing. Most image datasets are adopted to test correctness, overfitting, and robustness of ML systems.

Table 6 shows datasets related to natural language processing. The contents are usually text, sentences, or text files, applied to scenarios like robustness and correctness.

The datasets used to make decisions are introduced in Table 6. They are usually records with personal information, and thus are widely adopted to test the fairness of the ML models.

We also calculate how many datasets an ML testing paper usually uses in its evaluation (for those papers with an evaluation). Fig. 11 shows the results. Surprisingly, most papers use only one or two datasets in their evaluation; One reason might be training and testing machine learning models have high costs. There is one paper with as many as 600 datasets, but that paper used these datasets to evaluate data cleaning techniques, which has relatively low cost [179].

12. These tables do not list datasets adopted in data cleaning evaluation, because such studies usually involve hundreds of data sets [180]

TABLE 5
Datasets (1/4): Image Classification

Dataset	Description	Size	Usage
MNIST [253]	Images of handwritten digits	60,000+10,000	correctness, overfitting, robustness
Fashion MNIST [254]	MNIST-like dataset of fashion images	70,000	correctness, overfitting
CIFAR-10 [255]	General images with 10 classes	50,000+10,000	correctness, overfitting, robustness
ImageNet [256]	Visual recognition challenge dataset	14,197,122	correctness, robustness
IRIS flower [257]	The Iris flowers	150	overfitting
SVHN [258]	House numbers	73,257+26,032	correctness,robustness
Fruits 360 [259]	Dataset with 65,429 images of 95 fruits	65,429	correctness,robustness
Handwritten Letters [260]	Colour images of Russian letters	1,650	correctness,robustness
Balance Scale [261]	Psychological experimental results	625	overfitting
DSRC [262]	Wireless communications between vehicles and road side units	10,000	overfitting, robustness
Udacity challenge [77]	Udacity Self-Driving Car Challenge images	101,396+5,614	robustness
Nexar traffic light challenge [263]	Dashboard camera	18,659+500,000	robustness
MSCOCO [264]	Object recognition	160,000	correctness
Autopilot-TensorFlow [265]	Recorded to test the NVIDIA Dave model	45,568	robustness
KITTI [266]	Six different scenes captured by a VW Passat station wagon equipped with four video cameras	14,999	robustness

TABLE 6
Datasets (2/4): Natural Language Processing

Dataset	Description	Size	Usage
bAbI [267]	questions and answers for NLP	1000+1000	robustness
Tiny Shakespeare [268]	Samples from actual Shakespeare	100,000 character	correctness
Stack Exchange Data Dump [269]	Stack Overflow questions and answers	365 files	correctness
SNLI [270]	Stanford Natural Language Inference Corpus	570,000	robustness
MultiNLI [271]	Crowd-sourced collection of sentence pairs annotated with textual entailment information	433,000	robustness
DMV failure reports [272]	AV failure reports from 12 manufacturers in California[13]	keep updating	correctness

We also discuss research directions of building dataset and benchmarks for ML testing in Section 10.

9.5 Open-Source Tool Support in ML Testing

There are several tools specially designed for ML testing. Angell *et al*. presented Themis [213], an open-source tool for testing group discrimination.[14] There is also an ML testing framework for tensorflow, named *mltest*,[15] for writing simple ML unit tests. Similar to *mltest*, there is a testing framework for writing unit tests for pytorch-based ML systems, named *torchtest*.[16] Dolby *et al*. [237] extended WALA to enable static analysis for machine learning code using TensorFlow.

Compared to traditional testing, the existing tool support in ML testing is relatively immature. There remains plenty of space for tool-support improvement for ML testing.

10 Challenges and Opportunities

This section discusses the challenges (Section 10.1) and research opportunities in ML testing (Section 10.2).

10.1 Challenges in ML Testing

As this survey reveals, ML testing has experienced rapid recent growth. Nevertheless, ML testing remains at an early stage in its development, with many challenges and open questions lying ahead.

Challenges in Test Input Generation. Although a range of test input generation techniques have been proposed (see more in Section 5.1), test input generation remains challenging because of the large behaviour space of ML models.

13. The 12 AV manufacturers are: Bosch, Delphi Automotive, Google, Nissan, Mercedes- Benz, Tesla Motors, BMW, GM, Ford, Honda, Uber, and Volkswagen.
14. http://fairness.cs.umass.edu/
15. https://github.com/Thenerdstation/mltest
16. https://github.com/suriyadeepan/torchtest

TABLE 7
Datasets (3/4): Records for Decision Making

Dataset	Description	Size	Usage
German Credit [273]	Descriptions of customers with good and bad credit risks	1,000	fairness
Adult [274]	Census income	48,842	fairness
Bank Marketing [275]	Bank client subscription term deposit data	45,211	fairness
US Executions [276]	Records of every execution performed in the United States	1,437	fairness
Fraud Detection [277]	European Credit cards transactions	284,807	fairness
Berkeley Admissions Data [278]	Graduate school applications to the six largest departments at University of California, Berkeley in 1973	4,526	fairness
Broward County COMPAS [279]	Score to determine whether to release a defendant	18,610	fairness
MovieLens Datasets [280]	People's preferences for movies	100k-20m	fairness
Zestimate [281]	data about homes and Zillow's in-house price and predictions	2,990,000	correctness
FICO scores [282]	United States credit worthiness	301,536	fairness
Law school success [283]	Information concerning law students from 163 law schools in the United States	21,790	fairness

TABLE 8
Datasets (4/4): Others

Dataset	Description	Size	Usage
VirusTotal [284]	Malicious PDF files	5,000	robustness
Contagio [285]	Clean and malicious files	28,760	robustness
Drebin [286]	Applications from different malware families	123,453	robustness
Chess [287]	Chess game data: King+Rook versus King+Pawn on a7	3,196	correctness
Waveform [261]	CART book's generated waveform data	5,000	correctness

Search-based Software Test generation (SBST) [87] uses a meta-heuristic optimising search technique, such as a Genetic Algorithm, to automatically generate test inputs. It is a test generation technique that has been widely used in research (and deployment [288]) for traditional software testing paradigms. As well as generating test inputs for testing functional properties like program correctness, SBST has also been used to explore tensions in algorithmic fairness in requirement analysis. [205], [289]. SBST has been successfully applied in testing autonomous driving systems [245], [246], [247]. There exist many research opportunities in applying SBST on generating test inputs for testing other ML systems, since there is a clear apparent fit between SBST and ML; SBST adaptively searches for test inputs in large input spaces.

Existing test input generation techniques focus on generating adversarial inputs to test the robustness of an ML system. However, adversarial examples are often criticised because they do not represent real input data. Thus, an interesting research direction is to how to generate natural test inputs and how to automatically measure the naturalness of the generated inputs.

There has been work that tries to generate test inputs to be as natural as possible under the scenario of autonomous driving, such as DeepTest [76], DeepHunter [92] and DeepRoad [79], yet the generated images could still suffer from unnaturalness: sometimes even humans may not recognise the images generated by these tools. It is both interesting and challenging to explore whether such kinds of test data that are meaningless to humans should be adopted/valid in ML testing.

Challenges on Test Assessment Criteria. There has been a lot of work exploring how to assess the quality or adequacy of test data (see more in Section 5.3). However, there is still a lack of systematic evaluation about how different assessment metrics correlate, or how these assessment metrics correlate with tests' fault-revealing ability, a topic that has been widely studied in traditional software testing [290]. The relation between test assessment criteria and test sufficiency remains unclear. In addition, assessment criteria may provide a way of interpreting and understanding behaviours of ML models, which might be an interesting direction for further exploration.

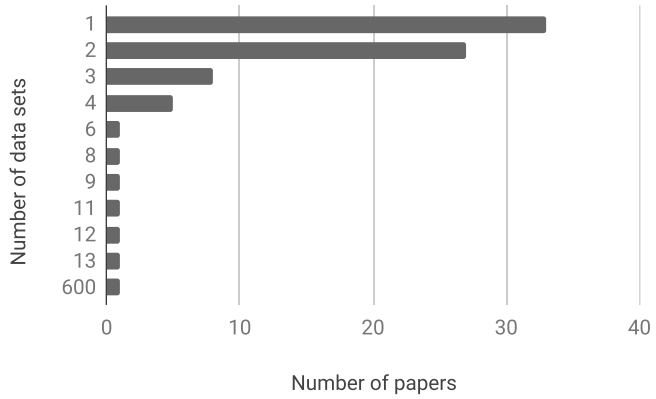


Fig. 11. Number of papers with different amounts of datasets in experiments

Challenges Relating to The Oracle Problem. The oracle problem remains a challenge in ML testing. Metamorphic relations are effective pseudo oracles but, in most cases, they need to be defined by human ingenuity . A remaining challenge is thus to automatically identify and construct reliable test oracles for ML testing.

Murphy *et al.* [128] discussed how flaky tests are likely to arise in metamorphic testing whenever floating point calculations are involved. Flaky test detection is a challenging problem in traditional software testing [288]. It is perhaps more challenging in ML testing because of the oracle problem.

Even without flaky tests, pseudo oracles may be inaccurate, leading to many false positives. There is, therefore, a need to explore how to yield more accurate test oracles and how to reduce the false positives among the reported issues. We could even use ML algorithm b to learn to detect false-positive oracles when testing ML algorithm a.

Challenges in Testing Cost Reduction. In traditional software testing, the cost problem remains a big problem, yielding many cost reduction techniques such as test selection, test prioritisation, and predicting test execution results. In ML testing, the cost problem could be more serious, especially when testing the ML component, because ML component testing usually requires model retraining or repeating of the prediction process. It may also require data generation to explore the enormous mode behaviour space.

A possible research direction for cost reduction is to represent an ML model as an intermediate state to make it easier for testing.

We could also apply traditional cost reduction techniques such as test prioritisation or minimisation to reduce the size of test cases without affecting the test correctness.

More ML solutions are deployed to diverse devices and platforms (e.g., mobile device, IoT edge device). Due to the resource limitation of a target device, how to effectively test ML model on diverse devices as well as the deployment process would be also a challenge.

10.2 Research Opportunities in ML testing

There remain many research opportunities in ML testing. These are not necessarily research challenges, but may greatly benefit machine learning developers and users as well as the whole research community.

Testing More Application Scenarios. Much current research focuses on supervised learning, in particular classification problems. More research is needed on problems associated with testing unsupervised and reinforcement learning.

The testing tasks currently tackled in the literature, primarily centre on image classification. There remain open exciting testing research opportunities in many other areas, such as speech recognition, natural language processing and agent/game play.

Testing More ML Categories and Tasks. We observed pronounced imbalance regarding the coverage of testing techniques for different machine learning categories and tasks, as demonstrated by Table 4. There are both challenges and research opportunities for testing unsupervised and reinforcement learning systems.

For instance, transfer learning, a topic gaining much recent interest, focuses on storing knowledge gained while solving one problem and applying it to a different but related problem [291]. Transfer learning testing is also important, yet poorly covered in the existing literature.

Testing Other Properties. From Fig. 10, we can see that most work tests robustness and correctness, while relatively few papers (less than 3 percent) study efficiency, model relevance, or interpretability.

Model relevance testing is challenging because the distribution of the future data is often unknown, while the capacity of many models is also unknown and hard to measure. It might be interesting to conduct empirical studies on the prevalence of poor model relevance among ML models as well as on the balance between poor model relevance and high security risks.

For testing efficiency, there is a need to test the efficiency at different levels such as the efficiency when switching among different platforms, machine learning frameworks, and hardware devices.

For testing property interpretability, existing approaches rely primarily on manual assessment, which checks whether humans could understand the logic or predictive results of an ML model. It will be also interesting to investigate the automatic assessment of interpretability and the detection of interpretability violations.

There is a lack of consensus regarding the definitions and understanding of fairness and interpretability. There is thus a need for clearer definitions, formalisation, and empirical studies under different contexts.

There has been a discussion that machine learning testing and traditional software testing may have different requirements in the assurance to be expected for different properties [292]. Therefore, more work is needed to explore and identify those properties that are most important for machine learning systems, and thus deserve more research and test effort.

Presenting More Testing Benchmarks. A large number of datasets have been adopted in the existing ML testing papers. As Tables 5, 6, 7, and 8 show, these datasets are usually those adopted for building machine learning systems. As far as we know, there are very few benchmarks like CleverHans[17] that are specially designed for ML testing research purposes, such as adversarial example construction.

More benchmarks are needed, that are specially designed for ML testing. For example, a repository of machine learning programs with real bugs would present a good benchmark for bug-fixing techniques. Such an ML testing repository, would play a similar (and equally-important) role to that played by data sets such as Defects4J[18] in traditional software testing.

Covering More Testing Activities. As far we know, requirement analysis for ML systems remains absent in the ML testing literature. As demonstrated by Finkelstein *et al.* [205], [289], a good requirements analysis may tackle many non-functional properties such as fairness.

Existing work is focused on off-line testing. Online-testing deserves more research efforts.

According to the work of Amershi *et al.* [8], data testing is especially important. This topic certainly deserves more research effort. Additionally, there are also many opportunities

17. https://github.com/tensorflow/cleverhans
18. https://github.com/rjust/defects4j

for regression testing, bug report analysis, and bug triage in ML testing.

Due to the black-box nature of machine learning algorithms, ML testing results are often more difficult for developers to understand, compared to traditional software testing. Visualisation of testing results might be particularly helpful in ML testing to help developers understand the bugs and help with the bug localisation and repair.

Mutating Investigation in Machine Learning System. There have been some studies discussing mutating machine learning code [128], [240], but no work has explored how to better design mutation operators for machine learning code so that the mutants could better simulate real-world machine learning bugs. This is another research opportunity.

11 Conclusion

We provided a comprehensive overview and analysis of research work on ML testing. The survey presented the definitions and current research status of different ML testing properties, testing components, and testing workflows. It also summarised the datasets used for experiments and the available open-source testing tools/frameworks, and analysed the research trends, directions, opportunities, and challenges in ML testing. We hope this survey will help software engineering and machine learning researchers to become familiar with the current status and open opportunities of and for of ML testing.

Acknowledgments

Jie M. Zhang and Mark Harman was supported by the ERC advanced grant with No. 741278. Lei Ma was supported by the JSPS KAKENHI Grant no.19K24348, 19H04086, Qdai-jump Research Program No.01277, and NVIDIA AI Tech Center (NVAITC). Yang Liu was supported by Singapore National Research Foundation with No. NRF2018NCR-NCR005-0001, National Satellite of Excellence in Trustworthy Software System No. NRF2018NCR-NSOE003-0001, NTU research grant NGF-2019-06-024. Before submitting, we sent the paper to those whom we cited, to check our comments for accuracy and omission. This also provided one final stage in the systematic trawling of the literature for relevant work. Many thanks to those members of the community who kindly provided comments and feedback on earlier drafts of this paper.

References

[1] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 1–18.

[2] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 2722–2730.

[3] G. Litjens *et al.*,"A survey on deep learning in medical image analysis," *Med. Image Anal.*, vol. 42, pp. 60–88, 2017.

[4] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[5] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: Testing software for discrimination," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 498–510.

[6] M. Pacula, "Unit-testing statistical software," 2011. [Online]. Available: http://blog.mpacula.com/2011/02/17/unit-testing-statistical-software/

[7] 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 *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2016, pp. 786–791.

[8] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *Proc. 41st Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2019, pp. 291–300.

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.

[10] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications," in *Proc. 19th Int. Conf. Softw. Eng. Knowl. Eng.*, 2007, Art. no 167.

[11] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proc. ACM 81 Conf.*, 1981, pp. 254–257.

[12] K. Androutsopoulos, D. Clark, H. Dan, M. Harman, and R. Hierons, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 573–583.

[13] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE Softw.*, vol. 12, no. 3, pp. 17–28, May 1995.

[14] D. Clark and R. M. Hierons, "Squeeziness: An information theoretic measure for avoiding fault masking," *Inf. Process. Lett.*, vol. 112, no. 8/9, pp. 335–340, 2012.

[15] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing (best paper award winner)," in *Proc. 8th Int. Work. Conf. Source Code Anal. Manipulation*, 2008, pp. 249–258.

[16] C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," in *Proc. Conf. Softw. Maintenance*, 1993, pp. 302–310.

[17] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[18] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar./Apr. 2010.

[19] A. M. Memon, "GUI testing: Pitfalls and process," *Computer*, vol. 35, no. 8, pp. 87–88, 2002.

[20] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263–272, 2005.

[21] G. Hains, A. Jakobsson, and Y. Khmelevsky, "Towards formal methods and software engineering for deep learning: Security, safety and productivity for DL systems development," in *Proc. Annu. IEEE Int. Syst. Conf.*, 2018, pp. 1–5.

[22] L. Ma *et al.*, "Secure deep learning engineering: A software quality assurance perspective," 2018, *arXiv: 1810.04538*.

[23] X. Huang *et al.*, "Safety and trustworthiness of deep neural networks: A survey," 2018, *arXiv: 1812.08342*.

[24] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa, "A survey of software quality for machine learning applications," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation Workshops*, 2018, pp. 279–284.

[25] F. Ishikawa, "Concepts in quality assessment for machine learning-from test data to arguments," in *Proc. Int. Conf. Conceptual Model.*, 2018, pp. 536–544.

[26] H. Braiek and F. Khomh, "On testing machine learning programs," 2018, *arXiv: 1812.02257*.

[27] J. Shawe-Taylor *et al.*, *Kernel Methods for Pattern Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[28] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. Cambridge, MA, USA: MIT Press, 2012.

[29] A. Paszke *et al.*, "Automatic differentiation in pytorch," 2017.

[30] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015," Software available from tensorflow.org.

[31] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[32] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://keras.io

[33] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.

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

[35] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.

[36] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied Linear Statistical Models*, vol. 4. Irwin Chicago, 1996.

[37] A. McCallum *et al.*, "A comparison of event models for naive bayes text classification," in *Proc. Workshop Learn. Text Categorization*, 1998, vol. 752, pp. 41–48.

[38] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015, Art. no. 436.

[39] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*.

[40] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 6645–6649.

[41] *IEEE Standard Classification for Software Anomalies*, IEEE Std 1044–2009 (Revision of IEEE Std 1044-1993), Jan. 2010.

[42] R. Werpachowski, A. György, and C. Szepesvári, "Detecting overfitting via adversarial examples," 2019, *arXiv: 1903.02380*.

[43] J. Cruz-Benito, A. Vázquez-Ingelmo, J. C. Sánchez-Prieto, R. Therón, F. J. García-Peñalvo, and M. Martín-González, "Enabling adaptability in web forms based on user characteristics detection through A/B testing and machine learning," *IEEE Access*, vol. 6, pp. 2251–2265, 2018.

[44] E. Kaufmann, O. Cappé, and A. Garivier, "On the complexity of best-arm identification in multi-armed bandit models," *The J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1–42, 2016.

[45] E. Breck, N. Polyzotis, S. Roy, S. Whang, and M. Zinkevich, "Data validation for machine learning," in *Proc. 2nd SysML Conf.*, 2019.

[46] C.-H. Cheng, G. Nührenberg, C.-H. Huang, and H. Yasuoka, "Towards dependability metrics for neural networks," 2018, *arXiv: 1806.02338*.

[47] S. Alfeld, X. Zhu, and P. Barford, "Data poisoning attacks against autoregressive models," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1452–1458.

[48] W. Qi, L. Tan, H. V. Pham, and T. Lutellier, "CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 1027–1038.

[49] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, vol. 5. Berlin, Germany: Springer, 2012.

[50] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, 2009.

[51] M. Kirk, *Thoughtful Machine Learning: A Test-Driven Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2014.

[52] V. Vapnik, E. Levin, and Y. L. Cun, "Measuring the VC-dimension of a learning machine," *Neural Comput.*, vol. 6, no. 5, pp. 851–876, 1994.

[53] D. S. Rosenberg and P. L. Bartlett, "The rademacher complexity of co-regularized kernel classes," in *Proc. 11th Int. Conf. Artif. Intell. Statist.*, 2007, pp. 396–403.

[54] J. Zhang, E. T. Barr, B. Guedj, M. Harman, and J. Shawe-Taylor, "Perturbed model validation: A new framework to validate model relevance," working paper or preprint, May 2019.

[55] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12–1990, Dec. 1990.

[56] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 1–17, 2013.

[57] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Proc. Int. Conf. Comput. Aided Verification*, 2017, pp. 97–117.

[58] C. Dwork, "Differential privacy," *Encyclopedia Cryptography Secur.*, pp. 338–340, 2011.

[59] P. Voigt and A. von dem Bussche, *The EU General Data Protection Regulation (GDPR): A Practical Guide*, 1st ed. Berlin, Germany: Springer, 2017.

[60] wikipedia, "California consumer privacy act," 2018. [Online]. Available: https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act

[61] R. Baeza-Yates and Z. Liaghat, "Quality-efficiency trade-offs in machine learning for text processing," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 897–904.

[62] S. Corbett-Davies and S. Goel, "The measure and mismeasure of fairness: A critical review of fair machine learning," 2018, *arXiv: 1808.00023*.

[63] D. S. Days III, "Feedback loop: The civil rights act of 1964 and its progeny," *Louis ULJ*, vol. 49, 2004, Art. no. 981.

[64] Z. C. Lipton, "The mythos of model interpretability," 2016, *arXiv:1606.03490*.

[65] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," 2017, *arXiv: 1702.08608*.

[66] T. Sellam, K. Lin, I. Y. Huang, M. Yang, C. Vondrick, and E. Wu, "DeepBase: Deep inspection of neural networks," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1117–1134.

[67] O. Biran and C. Cotton, "Explanation and justification in machine learning: A survey," in *Proc. Workshop Explainable AI*, 2017, Art. no. 8.

[68] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artif. Intell.*, vol. 267, pp. 1–38, 2018.

[69] B. Goodman and S. Flaxman, "European union regulations on algorithmic decision-making and a" right to explanation"," 2016, *arXiv:1606.08813*.

[70] C. Molnar, "Interpretable machine learning," 2019. [Online]. Available: https://christophm.github.io/interpretable-ml-book/

[71] J. Zhang *et al.*, "Search-based inference of polynomial metamorphic relations," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 701–712.

[72] M. O'Kelly, A. Sinha, H. Namkoong, R. Tedrake, and J. C. Duchi, "Scalable end-to-end autonomous vehicle testing via rare-event simulation," in *Proc. 32nd Neural Inf. Process. Syst.*, 2018, pp. 9827–9838.

[73] W. Xiang *et al.*, "Verification for machine learning, autonomy, and neural networks survey," 2018, *arXiv: 1810.01989*.

[74] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, Art. no. 38.

[75] A. Albarghouthi and S. Vinitsky, "Fairness-aware programming," in *Proc. Conf. Fairness Accountability Transparency*, 2019, pp. 211–219.

[76] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 303–314.

[77] udacity, "Udacity challenge," 2017. [Online]. Available: https://github.com/udacity/self-driving-car

[78] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 2672–2680.

[79] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 132–142.

[80] M.-Y. Liu, T. Breuel, and J. Kautz, "Unsupervised image-to-image translation networks," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 700–708.

[81] H. Zhou *et al.*, "DeepBillboard: Systematic physical-world testing of autonomous driving systems," 2018, *arXiv: 1812.10812*.

[82] X. Du, X. Xie, Y. Li, L. Ma, J. Zhao, and Y. Liu, "DeepCruiser: Automated guided testing for stateful deep learning systems," 2018, *arXiv: 1812.05339*.

[83] J. Ding, D. Zhang, and X.-H. Hu, "A framework for ensuring the quality of a big data service," in *Proc. IEEE Int. Conf. Services Comput.*, 2016, pp. 82–89.

[84] M. R. I. Rabin, K. Wang, and M. A. Alipour, "Testing neural program analyzers," 2019.

[85] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, 2019, Art. no. 40.

[86] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *Proc. Int. Conf. Softw. Eng.*, 2020.

[87] P. McMinn, "Search-based software test data generation: A survey," *Softw. Testing Verification Rel.*, vol. 14, no. 2, pp. 105–156, 2004.

[88] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1098–1105.

[89] A. Odena and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," 2018, *arXiv: 1807.10875*.

[90] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "DLFuzz: Differential fuzzing testing of deep learning systems," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2018, pp. 739–743.

[91] X. Xie *et al.*, "Coverage-guided fuzzing for deep neural networks," 2018, *arXiv: 1809.01266*.

[92] L. Ma *et al.*, "DeepGauge: Multi-granularity testing criteria for deep learning systems," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 120–131.

[93] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2018, pp. 408–426.

[94] J. Uesato *et al.*,, "Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures," in *Proc. Int. Conf. Learn. Representations*, 2019.

[95] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, 2019.

[96] S. Jha *et al.*, "ML-based fault injection for autonomous vehicles," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2019, pp. 112–124.

[97] S. Udeshi and S. Chattopadhyay, "Grammar based directed testing of machine learning systems," 2019, *arXiv: 1902.10027*.

[98] Y. Nie, Y. Wang, and M. Bansal, "Analyzing compositionality-sensitivity of NLI models," 2018, *arXiv: 1811.07033*.

[99] H. Wang, D. Sun, and E. P. Xing, "What if we simply swap the two text fragments? A straightforward yet effective way to test the robustness of methods to confounding signals in nature language inference tasks," 2018, *arXiv: 1809.02719*.

[100] A. Chan, L. Ma, F. Juefei-Xu, X. Xie, Y. Liu, and Y. S. Ong, "Metamorphic relation based adversarial attacks on differentiable neural computer," 2018, *arXiv: 1809.02444*.

[101] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 98–108.

[102] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *Proc. IEEE Intell. Vehicles Symp.*, 2018, pp. 1555–1562.

[103] A. Hartman, "Software and hardware testing using combinatorial covering suites, " in, *Graph Theory, Combinatorics and Algorithms*. Berlin, Germany: Springer, 2005, pp. 237–266.

[104] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[105] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[106] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.

[107] T. Chen, X.-S. Zhang, S.-Z. Guo, H.-Y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1758–1773, 2013.

[108] D. Gopinath, K. Wang, M. Zhang, C. S. Pasareanu, and S. Khurshid, "Symbolic execution for deep neural networks," 2018, *arXiv: 1807.10439*.

[109] A. Agarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Automated test generation to detect individual discrimination in ai models," 2018, *arXiv: 1809.03260*.

[110] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you?: Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 1135–1144.

[111] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 109–119.

[112] C. Murphy, G. Kaiser, and M. Arias, "Parameterizing random test data according to equivalence classes," in *Proc. 2nd Int. Workshop Random Testing: Co-Located 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 38–41.

[113] S. Nakajima and H. N. Bui, "Dataset coverage for testing machine learning computer programs," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf.*, 2016, pp. 297–304.

[114] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Hong Kong Univ. Sci. Technol., Hong Kong, Tech. Rep. HKUST-CS98–01, 1998.

[115] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *Proc. 20th Int. Conf. Softw. Eng. Knowl. Eng.*, 2008, pp. 867–872.

[116] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *Proc. 2nd Int. Workshop Metamorphic Testing*, 2017, pp. 28–34.

[117] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2009, pp. 436–445.

[118] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of metamorphic testing to supervised classifiers," in *Proc. 9th Int. Conf. Quality Softw.*, 2009, pp. 135–144.

[119] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[120] S. Nakajima, "Generalized oracle for testing machine learning computer programs," in *Proc. Int. Conf. Softw. Eng. Formal Methods*, 2017, pp. 174–179.

[121] A. Dwarakanath *et al.*, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 118–128.

[122] A. Sharma and H. Wehrheim, "Testing machine learning algorithms for balanced data usage," in *Proc. 12th IEEE Conf. Softw. Testing Validation Verification*, 2019, pp. 125–135.

[123] S. Al-Azani and J. Hassine, "Validation of machine learning classifiers using metamorphic testing and feature selection techniques," in *Proc. Int. Workshop Multi-Disciplinary Trends Artif. Intell.*, 2017, pp. 77–91.

[124] M. S. Ramanagopal, C. Anderson, R. Vasudevan, and M. Johnson-Roberson, "Failing to learn: Autonomously identifying perception failures for self-driving cars," *IEEE Robot. Autom. Lett.*, vol. 3, no. 4, pp. 3860–3867, Oct. 2018.

[125] X. Xie, Z. Zhang, T. Y. Chen, Y. Liu, P.-L. Poon, and B. Xu, "METTLE: A metamorphic testing approach to validating unsupervised machine learning methods," 2018.

[126] S. Nakajima, "Dataset diversity for metamorphic testing of machine learning software," in *Proc. Int. Workshop Structured Object-Oriented Formal Lang. Method*, 2018, pp. 21–38.

[127] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," 2018, *arXiv: 1808.08444*.

[128] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 189–200.

[129] L. Sun and Z. Q. Zhou, "Metamorphic testing for machine translations: MT4MT," in *Proc. 25th Australas. Softw. Eng. Conf.*, 2018, pp. 96–100.

[130] D. Pesu, Z. Q. Zhou, J. Zhen, and D. Towey, "A Monte Carlo method for metamorphic testing of machine translation services," in *Proc. 3rd Int. Workshop Metamorphic Testing*, 2018, pp. 38–45.

[131] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.

[132] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, pp. 216–226, 2014.

[133] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 785–796.

[134] A. Avizienis, "The methodology of N-version programming," *Softw. Fault Tolerance*, vol. 3, pp. 23–46, 1995.

[135] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Proc. Workshop Eng. Depend. Secure Mach. Learn. Syst.*, 2018, pp. 384–391.

[136] Y. Qin, H. Wang, C. Xu, X. Ma, and J. Lu, "SynEva: Evaluating ML programs by mirror program synthesis," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2018, pp. 171–182.

[137] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," 2017, *arXiv: 1711.07356*.

[138] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, "Fairness through awareness," in *Proc. 3rd Innovations Theoretical Comput. Sci. Conf.*, 2012, pp. 214–226.

[139] M. Hardt *et al.*, "Equality of opportunity in supervised learning," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 3315–3323.

[140] I. Zliobaite, "Fairness-aware machine learning: A perspective," 2017, *arXiv: 1708.00754*.

[141] B. Herman, "The promise and peril of human evaluation for model interpretability," 2017, *arXiv: 1711.07414*.

[142] D. Kang, D. Raghavan, P. Bailis, and M. Zaharia, "Model assertions for debugging machine learning," in *Proc. MLSys: Workshop Syst. ML Open Source Softw.*, 2018.

[143] L. Li and Q. Yang, "Lifelong machine learning test," in *Proc. Workshop "Beyond the Turing Test" AAAI Conf. Artif. Intell.*, 2015.

[144] J. Zhang *et al.*, "Predictive mutation testing," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 342–353.

[145] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," 2018, *arXiv: 1803.04792*.

[146] A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software," in *Proc. 19th Digit. Avionics Syst. Conf.*, 2000, vol. 1, pp. 1B6/1–1B6/7.

[147] J. Sekhon and C. Fleming, "Towards improved testing for deep learning," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: New Ideas Emerging Results*, 2019, pp. 85–88.

[148] L. Ma *et al.*, "Combinatorial testing for deep learning systems," 2018, *arXiv: 1806.07723*.

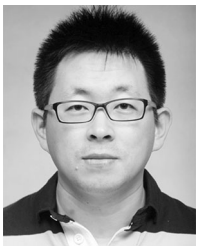
[149] L. Ma *et al.*, "DeepCT: Tomographic combinatorial testing for deep learning systems," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reengineering*, 2019, pp. 614–618.
[150] Z. Li, X. Ma, C. Xu, and C. Cao, "Structural coverage criteria for neural networks could be misleading," in *Proc. 41st Int. Conf. Softw. Eng.: New Ideas Emerging Results*, 2019, pp. 89–92.
[151] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.
[152] L. Ma *et al.*, "DeepMutation: Mutation testing of deep learning systems," 2018.
[153] W. Shen, J. Wan, and Z. Chen, "MuNN: Mutation analysis of neural networks," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur. Companion*, 2018, pp. 108–115.
[154] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 1123–1132.
[155] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," 2019, *arXiv: 1901.03768*.
[156] L. Zhang, X. Sun, Y. Li, Z. Zhang, and Y. Feng, "A noise-sensitivity-analysis-based test prioritization technique for deep neural networks," 2019, *arXiv: 1901.00054*.
[157] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lu, "Boosting operational DNN testing efficiency through conditioning," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 499–509.
[158] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," 2019.
[159] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng.*, 2012, pp. 271–280.
[160] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 129–140.
[161] S. S. Banerjee, S. Jha, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2018, pp. 586–597.
[162] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "MODE: Automated neural network model debugging via state differential analysis and input selection," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2018, pp. 175–186.
[163] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic, "Storm: Program reduction for testing and debugging probabilistic programming systems," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 729–739.
[164] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "TensorFlow debugger: Debugging dataflow graphs for machine learning," in *Proc. Reliable Mach. Learn. Wild-NIPS Workshop*, 2016.
[165] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia, "MISTIQUE: A system to store and query model intermediates for model diagnosis," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1285–1300.
[166] S. Krishnan and E. Wu, "PALM: Machine learning explanations for iterative debugging," in *Proc. 2nd Workshop Hum.-In-the-Loop Data Analytics*, 2017, Art. no. 4.
[167] B. Nushi, E. Kamar, E. Horvitz, and D. Kossmann, "On human intellect and machine failures: Troubleshooting integrative machine learning systems," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1017–1025.
[168] A. Albarghouthi, L. D'Antoni, and S. Drews, "Repairing decision-making programs under uncertainty," in *Proc. Int. Conf. Comput. Aided Verification*, 2017, pp. 181–200.
[169] W. Yang and T. Xie, "Telemade: A testing framework for learning-based malware detection systems," in *Proc. 32nd AAAI Conf. Artif. Intell. Workshops*, 2018, pp. 400–403.
[170] T. Dreossi, S. Ghosh, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Systematic testing of convolutional neural networks for autonomous driving," 2017, *arXiv: 1708.03309*.
[171] F. Tramer *et al.*, "FairTest: Discovering unwarranted associations in data-driven applications," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2017, pp. 401–416.
[172] Y. Nishi, S. Masuda, H. Ogawa, and K. Uetsuki, "A test architecture for machine learning product," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation Workshops*, 2018, pp. 273–278.
[173] P. S. Thomas, B. C. da Silva, A. G. Barto, S. Giguere, Y. Brun, and E. Brunskill, "Preventing undesirable behavior of intelligent machines," *Science*, vol. 366, no. 6468, pp. 999–1004, 2019.
[174] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. 14th Int. Joint Conf. Artif. Intell.*, 1995, vol. 14, pp. 1137–1145.
[175] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Boca Raton, FL, USA: CRC Press, 1994.
[176] N. Japkowicz, "Why question machine learning evaluation methods," in *Proc. AAAI Workshop Eval. Methods Mach. Learn.*, 2006, pp. 6–11.
[177] W. Chen, B. D. Gallas, and W. A. Yousef, "Classifier variability: Accounting for training and testing," *Pattern Recognit.*, vol. 45, no. 7, pp. 2661–2671, 2012.
[178] W. Chen, F. W. Samuelson, B. D. Gallas, L. Kang, B. Sahiner, and N. Petrick, "On the assessment of the added value of new predictive biomarkers," *BMC Med. Res. Methodol.*, vol. 13, no. 1, 2013, Art. no. 98.
[179] N. Hynes, D. Sculley, and M. Terry, "The data linter: Lightweight, automated sanity checking for ML data sets," 2017.
[180] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "BoostClean: Automated error detection and repair for machine learning," 2017, *arXiv: 1711.01299*.
[181] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 1781–1794, 2018.
[182] D. Baylor *et al.*, "TFX: A TensorFlow-based production-scale machine learning platform," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 1387–1395.
[183] D. M. Hawkins, "The problem of overfitting," *J. Chemical Inf. Comput. Sci.*, vol. 44, no. 1, pp. 1–12, 2004.
[184] H.-P. Chan, B. Sahiner, R. F. Wagner, and N. Petrick, "Classifier design for computer-aided diagnosis: Effects of finite sample size on the mean performance of classical and neural network classifiers," *Med. Phys.*, vol. 26, no. 12, pp. 2654–2668, 1999.
[185] B. Sahiner, H.-P. Chan, N. Petrick, R. F. Wagner, and L. Hadjiiski, "Feature selection and classifier performance in computer-aided diagnosis: The effect of finite sample size," *Med. Phys.*, vol. 27, no. 7, pp. 1509–1522, 2000.
[186] K. Fukunaga and R. R. Hayes, "Effects of sample size in classifier design," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 8, pp. 873–885, Aug. 1989.
[187] A. Gossmann, A. Pezeshk, and B. Sahiner, "Test data reuse for evaluation of adaptive machine learning algorithms: Over-fitting to a fixed'test'dataset and a potential solution," in *Proc. Med. Imag.: Image Perception Observer Perform. Technol. Assessment*, 2018, vol. 10577, Art. no. 105770K.
[188] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A simple and accurate method to fool deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2574–2582.
[189] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 2613–2621.
[190] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 39–57.
[191] W. Ruan, M. Wu, Y. Sun, X. Huang, D. Kroening, and M. Kwiatkowska, "Global robustness evaluation of deep neural networks with provable guarantees for L0 norm," 2018, *arXiv: 1804.05805*.
[192] D. Gopinath, G. Katz, C. S. Păsăreanu, and C. Barrett, "DeepSafe: A data-driven approach for assessing robustness of neural networks," in *Proc. Int. Symp. Autom. Technol. Verification Anal.*, 2018, pp. 3–19.
[193] R. Mangal, A. Nori, and A. Orso, "Robustness of neural networks: A probabilistic and practical perspective," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.: New Ideas Emerging Results*, 2019, pp. 93–96.
[194] S. S. Banerjee, J. Cyriac, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Towards a Bayesian approach for assessing faulttolerance of deep neural networks," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2019, pp. 25–26.
[195] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 582–597.
[196] N. Papernot, I. Goodfellow, R. Sheatsley, R. Feinman, and P. McDaniel, "cleverhans v1.0.0: An adversarial machine learning library," 2016, *arXiv:1610.00768*.
[197] N. Papernot *et al.*, "Technical report on the cleverhans v2.1.0 adversarial examples library," 2018, *arXiv:1610.00768*.
[198] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault injection for autonomous vehicles," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw. Workshops*, 2018, pp. 55–56.

[199] S. Jha *et al.*, "Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors," in *Proc. 3rd IEEE Int. Workshop Automotive Rel. Test*, 2018.
[200] H. Spieker and A. Gotlieb, "Towards testing of deep learning systems with training set reduction," 2019, *arXiv: 1901.04169*.
[201] S. Barocas and A. D. Selbst, "Big data's disparate impact," *Cal. L. Rev.*, vol. 104, 2016, Art. no. 671.
[202] P. Gajane and M. Pechenizkiy, "On formalizing fairness in prediction with machine learning," 2017, *arXiv: 1710.03184*.
[203] S. Verma and J. Rubin, "Fairness definitions explained," in *Proc. Int. Workshop Softw. Fairness*, 2018, pp. 1–7.
[204] N. Saxena, K. Huang, E. DeFilippis, G. Radanovic, D. Parkes, and Y. Liu, "How do fairness definitions fare? Examining public attitudes towards algorithmic definitions of fairness," 2018, *arXiv: 1811.03654*.
[205] A. Finkelstein, M. Harman, A. Mansouri, J. Ren, and Y. Zhang, "Fairness analysis in requirements assignments," in *Proc. 16th IEEE Int. Requirements Eng. Conf.*, 2008, pp. 115–124.
[206] M. J. Kusner, J. Loftus, C. Russell, and R. Silva, "Counterfactual fairness," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 4066–4076.
[207] N. Grgic-Hlaca, M. B. Zafar, K. P. Gummadi, and A. Weller, "The case for process fairness in learning: Feature selection for fair decision making," in *Proc. NIPS Symp. Mach. Learn. Law*, 2016, vol.1, p. 2.
[208] M. B. Zafar, I. Valera, M. G. Rodriguez, and K. P. Gummadi, "Fairness constraints: Mechanisms for fair classification," 2015, *arXiv:1507.05259*.
[209] B. Metevier *et al.*, "Offline contextual bandits with high probability fairness guarantees," in *Proc. 33rd Annu. Conf. Neural Inf. Process. Syst.*, 2019, pp. 14922–14933.
[210] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 661–670.
[211] P. Massart, "Concentration inequalities and model selection," 2007.
[212] A. Agarwal, A. Beygelzimer, M. Dudík, J. Langford, and H. Wallach, "A reductions approach to fair classification," 2018, *arXiv: 1803.02453*.
[213] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: Automatically testing software for discrimination," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 871–875.
[214] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: Finding defects' root causes," *CoRR*, 2018.
[215] S. A. Friedler, C. D. Roy, C. Scheidegger, and D. Slack, "Assessing the local interpretability of machine learning models," *CoRR*, 2019.
[216] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey, "Metamorphic relations for enhancing system understanding and use," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2018.2876433.
[217] W. Chen, B. Sahiner, F. Samuelson, A. Pezeshk, and N. Petrick, "Calibration of medical diagnostic classifier scores to the probability of disease," *Statist. Methods Med. Res.*, vol. 27, no. 5, pp. 1394–1409, 2018.
[218] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer, "Detecting violations of differential privacy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 475–489.
[219] B. Bichsel, T. Gehr, D. Drachsler-Cohen, P. Tsankov, and M. Vechev, "DD-Finder: Finding differential privacy violations by sampling and optimization," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 508–524.
[220] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data management challenges in production machine learning," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1723–1726.
[221] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," 2017, *arXiv: 1702.04267*.
[222] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 1245–1256.
[223] J. Wang, J. Sun, P. Zhang, and X. Wang, "Detecting adversarial samples for deep neural networks through mutation testing," *CoRR*, 2018.
[224] N. Carlini and D. Wagner, "Adversarial examples are not easily detected: Bypassing ten detection methods," in *Proc. 10th ACM Workshop Artif. Intell. Secur.*, 2017, pp. 3–14.
[225] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "ActiveClean: Interactive data cleaning for statistical modeling," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 948–959, 2016.
[226] S. Krishnan, M. J. Franklin, K. Goldberg, J. Wang, and E. Wu, "ActiveClean: An interactive data cleaning framework for modern machine learning," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 2117–2120.
[227] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proc. 8th IEEE Int. Conf. Data Mining*, 2008, pp. 413–422.
[228] S. Krishnan and E. Wu, "AlphaClean: Automatic generation of data cleaning pipelines," 2019, *arXiv: 1904.11827*.
[229] E. Rahm and H. H. Do, "Data cleaning: Problems and current approaches," *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, Dec. 2000.
[230] N. McClure, *TensorFlow Machine Learning Cookbook*. Birmingham, U.K.: Packt Publishing Ltd., 2017.
[231] T. Schaul, I. Antonoglou, and D. Silver, "Unit tests for stochastic optimization," in *Proc. Int. Conf. Learn. Representations*, 2014.
[232] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, "An empirical study on real bugs for machine learning programs," in *Proc. 24th Asia-Pacific Softw. Eng. Conf.*, 2017, pp. 348–357.
[233] L. Ma *et al.*, "An orchestrated empirical study on deep learning frameworks and platforms," 2018, *arXiv: 1811.05187*.
[234] Y. L. Karpov, L. E. Karpov, and Y. G. Smetanin, "Adaptation of general concepts of software testing to neural networks," *Program. Comput. Softw.*, vol. 44, no. 5, pp. 324–334, Sep. 2018.
[235] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 49–60.
[236] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 373–384.
[237] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: Analysis for machine learning programs," in *Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, 2018, pp. 1–10.
[238] Q. Xiao, K. Li, D. Zhang, and W. Xu, "Security risks in deep learning implementations," in *Proc. IEEE Secur. Privacy Workshops*, 2018, pp. 123–128.
[239] C. Roberts, "How to unit test machine learning code," 2017.
[240] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2018, pp. 313–324.
[241] 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," *J. Syst. Softw.*, vol. 84, no. 4, pp. 544–558, Apr. 2011.
[242] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Testing Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.
[243] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *Proc. Genetic Evol. Comput. Conf.*, 2004, pp. 1400–1412.
[244] M. Woehrle, C. Gladisch, and C. Heinzemann, "Open questions in testing of learned computer vision functions for automated driving," in *Proc. Int. Conf. Comput. Saf. Rel. Secur.*, 2019, pp. 333–345.
[245] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 1016–1026.
[246] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 63–74.
[247] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 143–154.
[248] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.
[249] W. Zheng *et al.*, "Testing untestable neural machine translation: An industrial case," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 314–315.
[250] W. Zheng *et al.*, "Testing untestable neural machine translation: An industrial case," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 314–315.
[251] W. Wang *et al.*, "Detecting failures of neural machine translation in the absence of reference translations," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.–Industry Track*, 2019, pp. 1–4.

[252] J. Burrell, "How the machine 'thinks': Understanding opacity in machine learning algorithms," *Big Data Soc.*, vol. 3, no. 1, 2016, Art. no. 2053951715622512.
[253] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.
[254] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," *CoRR*, 2017.
[255] A. Krizhevsky, V. Nair, and G. Hinton, Cifar-10 (Canadian Institute for Advanced Research) Tech. Rep.
[256] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[257] R. A. Fisher, "Iris data set," 1988. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/iris
[258] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," 2011.
[259] H. Mureşan and M. Oltean, "Fruit recognition from images using deep learning," *Acta Universitatis Sapientiae, Informatica*, vol. 10, pp. 26–42, Jun. 2018.
[260] O. Belitskaya, "Handwritten letters," 2019. [Online]. Available: https://www.kaggle.com/olgabelitskaya/handwritten-letters
[261] Balance scale data set, 1994. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/balance+scale
[262] S. Malebary, "DSRC vehicle communications data set," 2017. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/DSRC+Vehicle+Communications
[263] Nexar, "The nexar dataset," 2018. [Online]. Available: https://www.getnexar.com/challenge-1/
[264] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.
[265] X. Pan, Y. You, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," 2017, *arXiv: 1704.03952*.
[266] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The Int. J. Robot. Res.*, vol. 32, no. 11, pp. 1231–1237, 2013.
[267] Facebook research, "The babi dataset," 2015. [Online]. Available: https://research.fb.com/downloads/babi/
[268] A. Karpathy, J. Johnson, and L. Fei-Fei, "Visualizing and understanding recurrent networks," 2015, *arXiv:1506.02078*.
[269] Stack exchange data dump, 2014. [Online]. Available: https://archive.org/details/stackexchange
[270] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," 2015, *arXiv:1508.05326*.
[271] A. Williams, N. Nangia, and S. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2018, pp. 1112–1122.
[272] California DMV failure reports, 2019. [Online]. Available: https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2019
[273] Dr. H. Hofmann, "Statlog (german credit data) data set," 1994. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data)
[274] R. Kohav, "Adult data set," 1996. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/adult
[275] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decis. Support Syst.*, vol. 62, pp. 22–31, 2014.
[276] Executions in the United States, [Online]. Available: https://deathpenaltyinfo.org/views-executions
[277] A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi, "Calibrating probability with undersampling for unbalanced classification," in *Proc. IEEE Symp. Series Comput. Intell.*, 2015, pp. 159–166.
[278] P. J. Bickel, E. A. Hammel, and J. W. O'Connell, "Sex bias in graduate admissions: Data from Berkeley," *Science*, vol. 187, no. 4175, pp. 398–404, 1975.
[279] Propublica, "Data for the propublica story 'machine bias'," 2016. [Online]. Available: https://github.com/propublica/compas-analysis/
[280] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interactive Intell. Syst.*, vol. 5, no. 4, 2016, Art. no. 19.
[281] Zillow, "Zillow prize: Zillow's home value prediction (Zestimate)," 2016. [Online]. Available: https://www.kaggle.com/c/zillow-prize-1/overview
[282] U.S. Federal Reserve, "Report to the congress on credit scoring and its effects on the availability and affordability of credit," *Board of Governors of the Federal Reserve System*, 2007.
[283] Law School Admission Council, "LSAC national longitudinal bar passage study (NLBPS)," [Online]. Available: http://academic.udayton.edu/race/03justice/legaled/Legaled04.htm
[284] VirusTotal, "Virustotal," [Online]. Available: https://www.virustotal.com/#/home/search
[285] Contagio malware dump, 2013. [Online]. Available: http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-maliciou s-files.html
[286] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. E. R. T. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, vol. 14, pp. 23–26.
[287] A. Shapiro, "UCI chess (king-rook vs. king-pawn) data set," 1989. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Chess+%28King-Rook+vs.+King-Paw%n%29
[288] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proc. IEEE 18th Int. Work. Conf. Source Code Anal. Manipulation*, 2018, pp. 1–23.
[289] A. Finkelstein, M. Harman, A. Mansouri, J. Ren, and Y. Zhang, "A search based approach to fairness analysis in requirements assignments to aid negotiation, mediation and decision making," *Requirements Eng.*, vol. 14, no. 4, pp. 231–245, 2009.
[290] J. Zhang, L. Zhang, D. Hao, M. Wang, and L. Zhang, "Do pseudo test suites lead to inflated correlation in measuring test effectiveness?" in *Proc. 12th IEEE Conf. Softw. Testing Validation Verification*, 2019, pp. 252–263.
[291] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *J. Big data*, vol. 3, no. 1, 2016, Art. no. 9.
[292] S. Nakajima, "Quality assurance of machine learning software," in *Proc. IEEE 7th Global Conf. Consum. Electron.*, 2018, pp. 601–604.

Jie M. Zhang is a research associate in CREST, UCL, supervised by Earl Barr and Mark Harman. She has won the 2016 MSRA fellowship, the Top-ten Research Excellence Award of EECS, Peking University, Beijing, China, the 2015 National Scholarship, and so on. She is the co-chair of ASE SRC 2019 and has served on the program committees of ASE 2019 and Mutation 2019, 2018, and 2017. Her major research interests are software testing and program analysis.

Mark Harman is an engineering manager at Facebook London, where he manages a team, currently working on Search Based Software Engineering (SBSE). He is also a part time professor of software engineering with the Department of Computer Science, University College London, London, U. K., where he directed the CREST centre for 10 years (2006–2017) and was head of Software Systems Engineering (2012–2017). He is known for work on source code analysis, software testing, app store analysis, and empirical software engineering. He was the cofounder of the field SBSE, which has grown rapidly with more than 1,700 scientific publications from authors spread more than 40 countries. SBSE research and practice is now the primary focus of his current work in both the industrial and scientific communities.



Lei Ma received the BE degree from Shanghai Jiao Tong University, Shanghai, China, in 2009, and the ME and PhD degrees from The University of Tokyo, Tokyo, Japan, in 2011 and 2014, respectively. He is currently a tenured assistant professor with Information Science and Electrical Engineering, Kyushu University, Japan. His current research interests include the interdisciplinary fields of software engineering, security and trustworthy AI, with a special focus on proposing quality, reliability and security assurance solutions for machine learning engineering, and lifecycle.

Yang Liu received the bachelor's and PhD degrees from the National University of Singapore, Singapore, in 2005 and 2010, respectively. In 2012, he joined Nanyang Technological University, Singapore as a Nanyang assistant professor. He is currently a full professor, director of the cybersecurity lab, program director of HP-NTU Corporate Lab and deputy director of the National Satellite of Excellence of Singapore. In 2019, he received the University Leadership Forum chair professorship at NTU. He specialises in software verification, security, and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. By now, he has more than 280 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA fellowship, TRF fellowship, Nanyang assistant professor, Tan Chin Tuan fellowship, Nanyang Research Award 2019 and 10 best paper awards and one most influence system award in top software engineering conferences like ASE, FSE, and ICSE.