Machine Learning Testing: Survey, Landscapes and Horizons

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

Abstract—This paper provides a comprehensive survey of Machine Learning Testing (ML testing) research. It covers 128 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 recently 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% of the individuals in datasets.

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].

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 say [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. 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,

- *Jie M. Zhang and Mark Harman are with CREST, University College London, United Kingdom. Mark Harman is also with Facebook London. E-mail: jie.zhang, mark.harman@ucl.ac.uk*
- *Lei Ma is with Kyushu University, Japan. E-mail: malei@ait.kyushu-u.ac.jp*
- *Yang Liu is with School of Computer Science and Engineering, Nanyang Technological University. E-mail: yangliu@ntu.edu.sg*

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: Figure 1 shows the cumulative number of publications on the topic of testing machine learning systems between 2007 and June 2019. From this figure, we can see that 85% of papers have appeared since 2016, testifying to the emergence of new software testing domain of interest: machine learning testing.

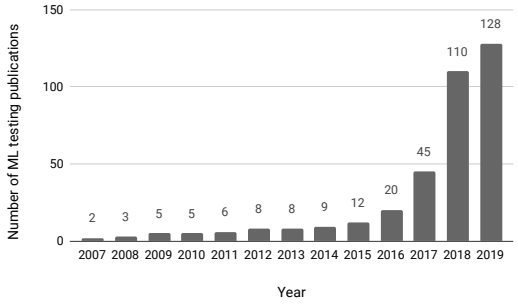

Figure 1: Machine Learning Testing Publications (accumulative) during 2007-2019

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 GUIs (Graphical User Interfaces) 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 as one aspect of guarantee 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 128 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 128 papers we collected, over 110 of them tackle supervised learning testing, three of them tackle unsupervised learning testing, and only one paper tests reinforcement learning. Additionally, most of them (85) centre around 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.

Figure 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.

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.

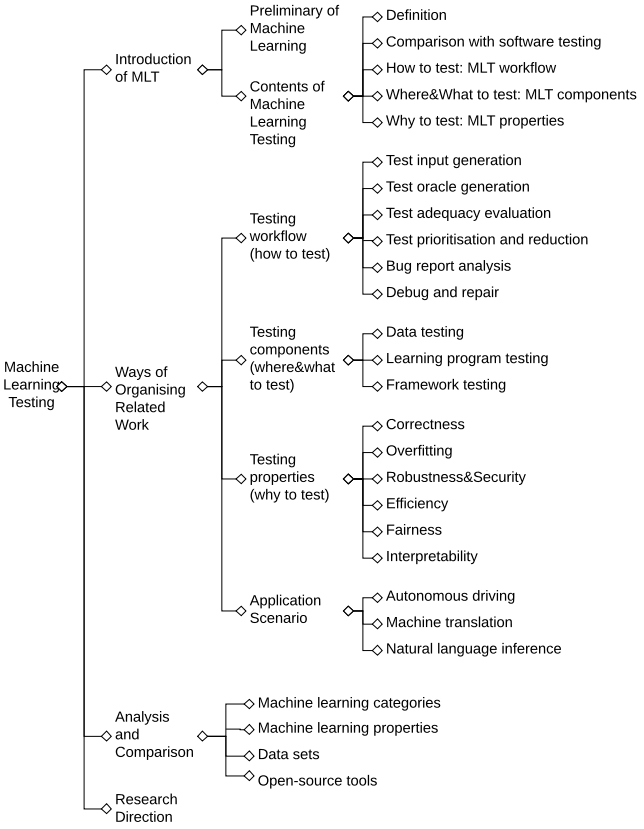

Figure 2: Tree structure of the contents in this paper

- **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 $\mathcal{X} = (x_1, ..., x_m)$ be the set of unlabelled training data. Let $\mathcal{Y} = (c(x_1), ..., c(x_m))$ 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 requires a set of states S, actions A, a transition probability and a reward probability. It learns how to take actions from A under different S to get the best reward.

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

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.

Figure 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 player to maximise the rewards for a game agent.

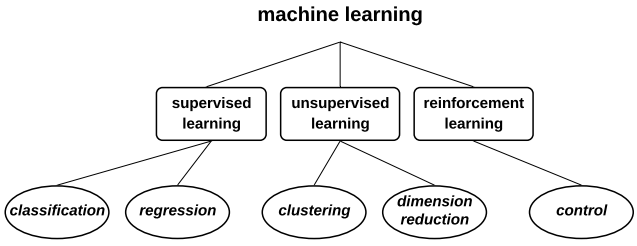

Figure 3: Machine learning categories and tasks

In addition, machine learning can be classified into **classic machine learning** and **deep learning**. Algorithms like Decision 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 (**why** to test), and testing components (**where** and **what** 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[1].

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.

Having defined ML bugs, in this paper, we define ML testing as the activities aimed to detect ML bugs.

Definition 2 (ML Testing). Machine Learning Testing (ML testing) refers to any activities 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 in traditional software testing, due to the fact that not only the code may contain bugs, but also the data. When we try to detect the bugs in data, one may even use a toy training program as a test input to check some must-to-hold data properties.

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.

[1] 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].

3.2.1 Role of Testing in ML Development

Figure 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 fail 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 a ML testing workflow adapted from classic software testing workflow. Figure 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 of Figure 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, tests inputs are either sampled from the collected data or generated somehow 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) 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 some evaluation metrics to check the quality of tests, i.e., the ability of the tests to expose ML problems.

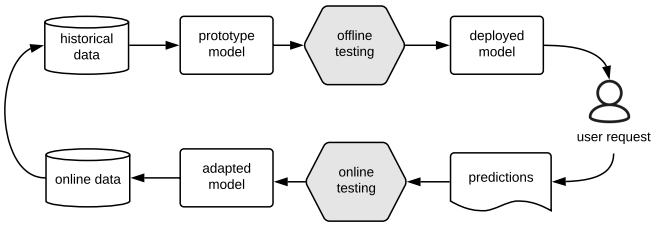

Figure 4: Role of ML testing in ML system development

The test execution results yield a bug report to help developers 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

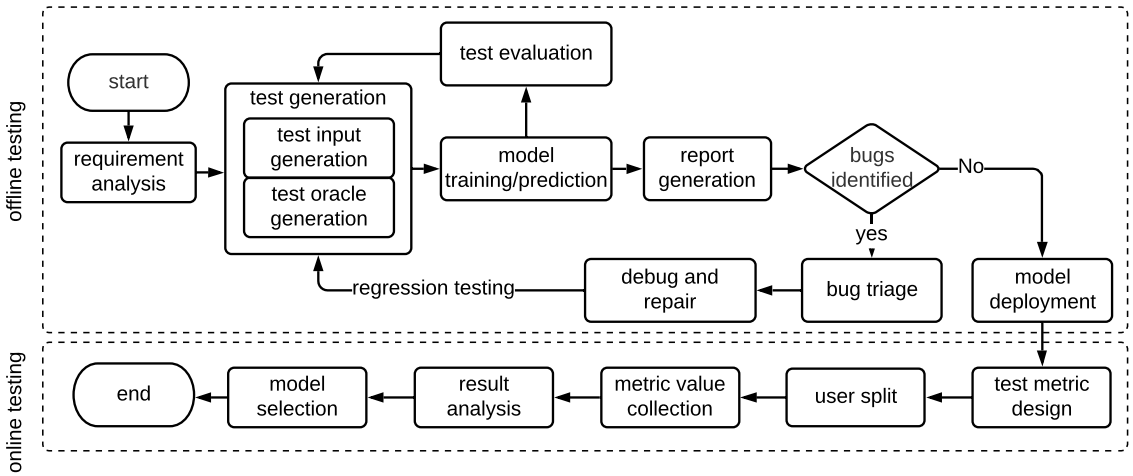

Figure 5: Workflow of ML testing

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.

The workflow of online testing is shown by the bottom of Figure 5. Usually the users are split into different groups to conduct control experiments, to help find out which model is better, or whether the new model is superior to the old model under certain application contexts.

A/B testing is one of the key types of online testing of machine learning to validate the deployed models [43]. It is a splitting testing technique to compare two versions of the systems (e.g., web pages) that involve customers. 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.

MRB (Multi-Rrmed 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.

Figure 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; Learning program is the code for running to train the model; 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 in the following.

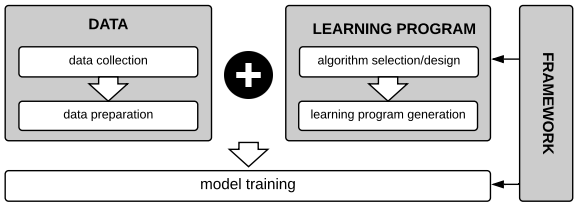

Figure 6: Components (shown by the grey boxes) involved in ML model building.

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.

Bug Detection in Frameworks. Machine Learning requires a lot of computations. As shown by Figure 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 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 **why** to test in ML testing: for what conditions ML testing needs to guarantee for a trained model. This section lists some properties that the literature concern the most, including basic functional requirements (i.e., correctness and overfitting degree) and non-functional requirements (i.e., efficiency, robustness[2], fairness, interpretability).

These properties are not strictly independent with each other when considering the root causes, yet they are different external performances 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 $\mathcal{X} = (x_1, ..., x_m)$ be the set of unlabelled test data sampled from $\mathcal{D}$. Let h be the machine learning model under test. Let $\mathcal{Y}' = (h(x_1), ..., h(x_m))$ be the set of predicted labels corresponding to each training item x_i. Let $\mathcal{Y} = (y_1, ..., y_m)$ 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.

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

3.4.2 Overfitting Degree

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 should be no more complex than just needed [51]. Otherwise, the model may fail to have good performance and generalise to new data.

We define the problem of detecting whether the machine learning algorithm's capacity fits the data as the identification of *Overfitting Degree*. The capacity of the algorithm is usually approximated by VC-dimension [52] or Rademacher Complexity [53] for classification tasks.

Definition 5 (Overfitting Degree). Let $\mathcal{D}$ be the training data distribution. Let $R(\mathcal{D}, \mathcal{A})$ be the simplest required capacity of machine learning algorithm $\mathcal{A}$. $R'(\mathcal{D}, \mathcal{A}')$ is the capacity of the machine learning algorithm $\mathcal{A}'$ under test. The overfitting degree is the difference between $R(\mathcal{D}, \mathcal{A})$ and $R'(\mathcal{D}, \mathcal{A}')$.

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

The overfitting degree aims to measure how much a machine learning algorithm fails to fit future data or predict future observations reliably because it is 'overfitted' to currently available training data.

The quantitative overfitting analysis of machine learning is a challenging problem [42]. 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 [54], [55] as: 'The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions'. Taking the similar spirit of 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 $E(\delta(S))$:

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

Robustness thus measures the resilience of an ML system towards perturbations.

A popular sub-category of robustness is called *adversarial robustness*. For adversarial robustness, the perturbations are designed to be hard to detect. Referring to the work of Katz et al. [56], 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'

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

Local adversarial robustness concerns the robustness at one specific test input, while global adversarial robustness requests 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'

$$\forall x, x' : ||x - x'||_p \leq \delta \rightarrow h(x) - h(x') \leq \epsilon \quad (6)$$

3.4.4 Security

Definition 9 (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 be easy to suffer from adversarial attacks (i.e., fooling the ML system via generating adversarial examples, which are special test inputs modified from original inputs but look the same as original inputs to the humans); if it is less robust in resisting the perturbations in the training data, the system may be easy to suffer from data poisoning (i.e., change the predictive behaviour via modifying the training data).

Nevertheless, low robustness is just one cause for high-security risk. 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

Machine learning is widely adopted to predict individual habits or behaviours to maximise the benefits of many companies. For example, Netflix offered a large dataset (with the ratings of around 500,000 users) for an open competition to predict user ratings for films. Nevertheless, sensitive information about individuals in the training data can be easily leaked. Even if the data is anonymised (i.e., to encrypt or remove personally identifiable information from datasets to keep the individual anonymous), there is a high risk of linkage attack, which means the individual information may still be recovered to some extent via linking the data instances among different datasets.

We define privacy in machine learning as 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 [57].

Definition 10 (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

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

In other words, ϵ-Differential privacy ensures that the learner should not get much more information from D_1 than D_2 when D_1 than D_2 differs in only one instance.

Data privacy has been regulated by The EU General Data Protection Regulation (GDPR) [58] and California CCPA [59], make the protection of data privacy a hot research topic. Nevertheless, the current research mainly focus on data privacy is 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 [60]. For example, given a large VGG-19 model which hundreds of MB, to deploy such a model to a mobile device, certain optimisation, compression, and device-oriented customisation must be performed to make it feasible for a mobile device execution in a reasonable time, but the accuracy is often sacrificed.

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. It learns what human beings teach (i.e., in form of training data), while human beings may have bias over cognition, further affecting the data collected or labelled and the algorithm designed, leading to bias issues. It is thus important to ensure that the decisions made by a machine learning system are in the right way and for the right reason, to avoid problems in human rights, discrimination, law, and other ethical issues.

The characteristics that are sensitive and need to be protected against unfairness are called *protected characteristics* [61] 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[3].

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 in details.

[3] To prohibit discrimination 'in a place of public accommodation on the basis of sexual orientation, gender identity, or gender expression' [62].

3.4.8 Interpretability

Machine learning models are oftentimes applied to assist/-make decisions in medical treatment, income prediction, or personal credit assessment. It is essential for human beings to understand the logic and reason behind the final decisions made by some certain machine learning system, so that human beings can build more trust over the decisions made by ML to make it more socially acceptable, understand the cause of decisions to avoid discrimination and get more knowledge, transfer the models to other situations, and avoid safety risks as much as possible [63], [64], [65].

The motives and definitions of interpretability are diverse and still discordant [63]. Nevertheless, unlike fairness, a mathematical definition of ML interpretability remains elusive [64]. Referring to the work of Biran and Cotton [66] as well as the work of Miller [67], we define the interpretability of ML as follows.

Definition 11 (Interpretability). ML Interpretability refers to 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) [63]. Interpretability is also regarded as a request by regulations like GDPR [68], 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 [69].

3.5 Software Testing vs. 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 playing 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 update of training data.

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 various forms and one may need to use a piece of code as a test input to test the data. Note that we separate the definition of 'test input' and 'test data'. Test inputs in ML testing could be but 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, yet the answers are usually unknown. There would be no need to write such programs, if the correct answer were known' as described by Davis and Weyuker [11]. Thus, the oracles that could be adopted by ML testing are more difficult to obtain and are usually some pseudo oracles such as metamorphic relations [70].

5) Test adequacy criteria: test adequacy criteria are used to provide quantitative measurement on the degree of the target software is tested. Up to present, many adequacy criteria are proposed and widely adopted in software industry, e.g., line coverage, branch coverage, dataflow coverage. However, due to the fundamental difference of programming paradigm and logic representation format for machine learning software and traditional software, new test adequacy criteria are required so as 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 tend 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.

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	unknown
Adequacy Criteria	coverage/mutation score	unknown
False positives in bugs	rare	prevalent
Tester	developer	data scientist, algorithm designer, developer

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, overfitting degree, 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 March 8, 2019.

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

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 [71] 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 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 19 papers when dealing with the author feedback. Altogether, we collected 128 papers.

Figure 7 shows the distribution of papers published in different research venues. Among all the papers, 37.5% papers are published in software engineering venues such as ICSE, FSE, ASE, ICST, and ISSTA; 11.7% papers are published in systems and network venues; surprisingly, only 13.3% of the total papers are published in artificial intelligence venues such as AAAI, CVPR, and ICLR. Additionally, 25.0% of the papers have not yet been published via peer-reviewed venues (the arXiv part). This distribution of different venues indicates that ML testing is the most widely published in software engineering venues, but with a wide publishing venue diversity.

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	-	-	19
Overall	-	-	128

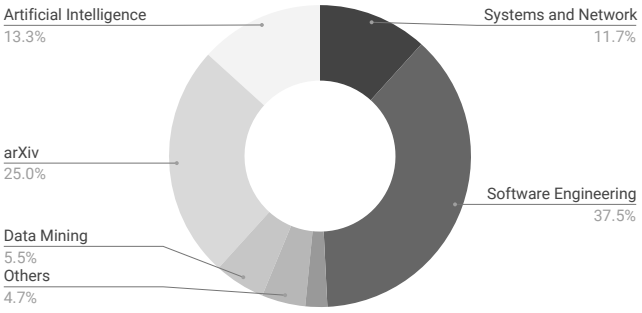

Figure 7: Publication Venue Distribution

4.4 Paper Organisation

We present the literature review from two aspects: 1) a literature review of the collected papers, 2) a statistical analysis of the collected papers, datasets, and tools, The sections and the corresponding contents are presented in Table 3.

Table 3: Review Schema

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

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 overfitting degree 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.

We'd like to strengthen that 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). That is to say, for one ML testing paper, it may fit multiple aspects if being viewed from different angles. For such a kind of paper, we fit it into one or more sections wherever applicable.

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 of presenting the 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 contents are presented in Section 10.

5 ML Testing Workflow

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

ML testing includes offline testing and online testing, the current research centres around offline testing. 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 they adopt.

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% more neurons than the same number of randomly picked inputs and adversarial inputs.

To create useful and effective data for autonomous driving systems, *DeepTest* [72] 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[4]: linear, affine, and convolutional. The evaluation of DeepTest uses the Udacity self-driving car challenge dataset [73]. It detected more than 1,000 erroneous behaviours on CNNs and RNNs with low false positive rates[5].

Generative adversarial networks (GANs) [74] 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. [75] applied GAN to deliver driving scene-based test generation with various weather conditions. They

[4] https://github.com/itseez/opencv(2015)

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

sampled images from Udacity Challenge dataset [73] and YouTube videos (snowy or rainy scenes), and fed them into UNIT framework[6] for training. The trained model takes the whole Udacity images as the seed inputs and yield transformed images as generated tests.

Zhou et al. [77] 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. [78] 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. [79] 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 adding new or increasing the number/shape of artificial mitochondrion into the biological cell images, which can arouse easy-to-identify changes in the classification results.

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], [80], [81]. These two techniques have also been proved to be effective in exploring the input space of ML testing:

Odena et al. [82] 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. [83], 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, but only need to try minimum changes over the original inputs to find those new inputs that improve neural coverage but have different predictive results with the original inputs. The preliminary evaluation on MNIST and ImageNet shows that compared with DeepXplore, DLFuzz is able to generate 135% to 584.62% more inputs with 20.11% less time consumption.

Xie et al. [84] presented a metamorphic transformation based coverage guided fuzzing technique, DeepHunter, which leverages both neuron coverage and coverage criteria presented by DeepGauge [85]. 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 achieve high coverage and bug detection capability.

Wicker et al. [86] 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. [87] proposed to evaluate reinforcement learning with adversarial example generation. The detection of catastrophic failures is expensive because failures are rare. To alleviate the cost problem, 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. [88] combined fuzzing and metamorphic testing to test the LiDAR obstacle-perception module of real-life self-driving cars, and reported previously unknown fatal software faults eight days before Uber's deadly crash in Tempe, AZ, in March 2018. Jha et al. [89] 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 evaluation based on two production-grade AV systems from NVIDIA and Baidu revealed many situations where faults lead to safety violations.

Udeshi and Chattopadhyay [90] generates inputs for text classification tasks and make fuzzing considering the grammar under test as well as the distance between inputs. Nie et al. [91] and Wang et al. [92] mutated the sentences in NLI (Natural Language Inference) tasks to generate test inputs for robustness testing. Chan et al. [93] generated adversarial examples for DNC to expose its robustness problems. Udeshi et al. [94] 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. [95] proposed a framework for testing autonomous driving systems. In their work they compared three test generation strategies: random fuzz test generation, covering array [96] + fuzz test generation, and covering array + search-based test generation (using Simulated Annealing algorithm [97]). 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 [98]. 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

[6] A recent DNN-based method to perform image-to-image transformation [76]

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 [99]. DSE has been found to be accurate and effective, and has been the fundamental technique of some vulnerability discovery tools [100].

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 abstract the data 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 to find errors in ML systems. 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 comes many challenges. Gopinath [101] 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 structure of the ML models are usually very complex and are beyond the capabilities of current symbolic reasoning tools.

Considering these challenges, Gopinath [101] 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. [102] apply LIME [103], 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 random test generation approach THEMIS [5].

Sun et al. [104] 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, and demonstrates yields over 10% higher neuron coverage than DeepXplore for the evaluated models.

5.1.4 Synthetic Data to Test Learning Program

Murphy et al. [105] 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. [106] used synthetic data with known distributions to test overfitting. Nakajima and Bui [107] 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, to enable the judgement of bug existence. 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 have been proposed by Chen et al. [108] 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 granularity 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 against each piece of instance, 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. [109] 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 2008 introduction, 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. [110] 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 the classification accuracy, such as adding 10% training images into each category of the training data set or removing one category of the data from the dataset. The evaluation is based on a classification of biology cell images.

Murphy et al. [111] presented function-level metamorphic relations. The evaluation on 9 machine learning applications indicated that functional-level properties were 170% more effective than application-level properties.

Fine-grained Data Transformation. The metamorphic relations of Murphy et al. [109] introduced above are general relations suitable for both supervised and unsupervised learning algorithms. In 2009, Xie et al. [112] 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 [113] indicates that not all metamorphic relations are proper or necessary. The differences in the metamorphic relations between SVM and neural networks are also discussed in [114]. Dwarakanath et al. [115] 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% of the injected bugs. Sharma and Wehrheim [116] 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. [106] 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 overfitting degree is.

Al-Azani and Hassine [117] 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. [72] and Zhang et al. [75] 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. [118] 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. [119] 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. [107], [120] 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. [121] and Breck et al. [45] studied the metamorphic relation between training data and new data. If the training data and new data have different distributions, the training data have problems. 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. [122] implemented a framework called Amsterdam to automates the process of using metamorphic relation to detect ML bugs. The framework reduces false positives via setting thresholds when doing result comparison. They also developed Corduroy [111], which extended Java Modelling Language to let developers specify metamorphic properties and generate test cases for ML testing.

We will introduce more related work about using domain-specific metamorphic relations of testing autonomous driving, Differentiable Neural Computer (DNC) [93], machine translation systems [123], [124], biological cell classification [79], and audio-based deep learning systems [78] in Section 8.

5.2.2 Cross-Referencing as Test Oracles

Cross-Referencing is another type of test oracles in 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], [125]. Differential testing is one of the major testing oracles for detecting compiler bugs [126]. It is closely related with N-version programming [127]: 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 implementation of an algorithm yields

different outputs on one identical input, then one or both of the implementation contains a defect. Alebiosu et al. [128] evaluated this idea on machine learning, and successfully found 5 faults from 10 Naive Bayes implementations and 4 faults from 20 k-nearest neighbour implementation.

Pham et al. [48] also adopted multiple implementation 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 [83] used differential testing as test oracles to find effective test inputs. Those test inputs causing different behaviours among different algorithms or models are preferred during test generation.

Most differential testing relies on multiple implementations or versions, while Qin et al. [129] 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.

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 [130], fairness [131], [132], [133], and interpretability [64], [134]. These measurements are not direct oracles for testing, but are essential for testers to understand and evaluate the property under test, and provide some actual statistics that can be compared with the expected ones. For example, the definitions of different types of fairness [131], [132], [133] (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. [135] 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 [136]. 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 is to find out whether the existing tests have a good fault-revealing ability, and provides an objective confidence measurement on the 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 [137]. 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 are learned from training data. For example, in the study of Pei et al. [1], 100 % 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] pioneered to propose the very 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. [85] extended the concept of neuron coverage. They first profile a DNN based on the training data, so that obtain the activation behaviour of each neuron against the training data. Based on this they propose more fined-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. [138] proposed four test coverage criteria that are tailored to the distinct features of DNN inspired by the MC/DC coverage criteria [139]. 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 [140].

Layer-level coverage. Ma et al. [85] 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 [141], [142] , 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 [140] 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 the stateful machine learning system like recurrent neural network (RNN). The RNN-based ML system achieves great success in application to handle sequential inputs, e.g., speech audio, natural language, cyber physical control signals. Towards analyzing such stateful ML systems, Du et al. [78] 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 [140] examined the existing testing methods for DNNs and discussed the limitations of these criteria.

Up to present, most proposed coverage criteria are based on the structure of a DNN. Li et al. [143] 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 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 [137], [144]. 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. [145] 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. [146] 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 with structural coverage criteria, mutation testing based criteria is 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. [121] introduced **surprise adequacy** to measure the coverage of discretised input surprise range for deep learning systems. They argued that 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).

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 to examine. Breck et al. [147] 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 work 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. They also mentioned that to randomly generate input data and train the model for a single step of gradient descent is quite powerful for detecting library mistakes. 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. [148] used DNN metrics like cross entropy, surprisal, and Bayesian uncertainty to prioritise test inputs and 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. [149] 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 noises, because the instance that is more sensitive to noise is more likely to yield adversarial examples.

Li et. al [150] 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 pretrained models with three image datasets: MNIST, Udacity challenge, and ImageNet. The results show that compared with random sampling, their approach samples only half the test inputs to achieve a similar level of precision.

5.5 Bug Report Analysis

Thung et al. [151] 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 most ML bugs, i.e., 22.6% 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% of bugs are non-functional bugs. 5.6% of bugs are data bugs.

Zhang et al. [152] 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%), unaligned tensor (13.7%), and incorrect model parameter or structure (21.7%)

Banerjee et al. [153] analysed the bug reports of autonomous driving systems from 12 autonomous vehicle manufacturers that drove a cumulative total of 1,116,605 VC miles in California. They used NLP technologies to classify the causes of disengagements (i.e., failures that cause the control of the vehicle to switch from the software to the human driver) into 10 types. The issues in machine learning systems and decision control account for the primary cause of 64 % of all disengagements based on their report analysis.

5.6 Debug and Repair

Data Resampling. The generated test inputs introduced in Section 5.1 can not only expose ML bugs, but are also studied to serve as a part of the training data and improve the model's correctness through retraining. For example, DeepXplore achieves up to 3% improvement in classification accuracy by retraining a deep learning model on generated inputs. DeepTest [72] improves the model's accuracy by 46%.

Ma et al. [154] 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. Cai et al. [155] 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. [156] proposed the MISTIQUE system to capture, store, and query model intermediates to help the debug. Krishnan and Wu [157] 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 what training data impact the prediction the most, and 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. [158] 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.

5.7 General Testing Framework and Tools

There has also been some 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 [159]. Dreossi et al. [160] 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. [161] proposed a comprehensive testing tool to help developers to test and debug fairness bugs with an easily interpretable bug report. Nishi et al. [162] 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.

6 ML Properties to be Tested

ML properties concern what required conditions we should care about in 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 do cross-validation. In hold out cross-validation [163], the data are split into two parts: one part as the training data and the other part as test data[7]. 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 [164], and thus the test data may contain repeated instances.

[7] Sometimes validation set is also needed to help train the model, which circumstance the validation set will be isolated from the training set.

There are several widely-adopted correctness measurements such as accuracy, precision, recall, and AUC. There has been work [165] 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 very unbalanced. An indication of these work is that we should carefully choose the performance metrics and it is essential to consider their meanings when adopting them. Chen et al. studied the variability of both training data and test data when assessing the correctness of an ML classifier [166]. 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 F-test has the best performance [167].

To better capture correctness problems, Qin et al. [129] 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 on the popularity of prevalence of correctness problems among all the reported ML bugs: Zhang et al. [152] studied 175 Tensorflow bug reports from StackOverflow QA (Question and Answer) 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 [168], [169], [170] (see more in Section 7.1), test input or test oracle design [110], [114], [115], [117], [124], [147], [154], and test tool design [156], [158], [171] (see more in Section 5).

6.2 Overfitting

When a model is too complex for the data, even the noise of training data is fitted by the model [172]. The overfitting easily happens, especially when the training data is not sufficient, [173], [174], [175]. Overfitting may lead to high correctness on the existing training data yet low correctness on the unseen data.

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 could be unlikely to detect overfitting if the test data is unrepresentative of potential unseen data.

Zhang et al. [106] introduced Perturbed Model Validation (PMV) to detect overfitting (and also underfitting). 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 much better performance and provides more recognisable signal in detecting both overfitting/underfitting than 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. DeepMind presents an overfitting detection approach via generating adversarial examples from test data [42]. 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. [176] 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 of an ML model; it is better to choose the algorithm with equal correctness but relatively small training time.

Ma et al. [154] treated data as one of the important reasons for the low performance of the model and its overfitting. They tried to relieve the overfitting problem via re-sampling the training data. The approach is found to improve test accuracy from 75% to 93% in average based on evaluation of 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 [130]; a robust system should maintain performance in the presence of noise.

Moosavi-Dezfooli et al. [177] proposed DeepFool that computes perturbations (added noise) that 'fool' deep networks so as to quantify their robustness. Bastani et al. [178] 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 [179] created a set of attacks that can be used to construct an upper bound on the robustness of a neural network. Tjeng et al. [130] proposed to use the distance between a test input and its closest adversarial example to measure the robustness. Ruan et al. [180] provided global robustness lower and upper bounds based on the test data to quantify the robustness. Gopinath [181] et al. 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. [182] 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. [183] 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, but 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 [179] 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 [184].

Adversarial input generation has been widely adopted to test the robustness of autonomous driving system [1], [72], [75], [85], [86]. There has also been research efforts in generating adversarial inputs for NLI models [91], [92](Section 8.3), malware detection [159], and Differentiable Neural Computer (DNC) [93].

Papernot et al. [185], [186] designed a library to standardise the implementation of adversarial example construction. They pointed out that standardising adversarial example generation is very 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 to check the robustness of neural networks include symbolic execution [101], [104], fuzz testing [83], combinatorial Testing [141], and abstract interpretation [182]. In Section 5.1, we introduce more contents about test generation techniques.

6.3.3 Perturbation Targeting the Whole System

There has been research on generating perturbations in the system and record the system's reactions towards them. Jha et al. [187] 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 [188], 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 [89], further presented by Jha et al., is a fault-injection engine that mines situations and faults that maximally impact AV safety.

Tuncali et al. [95] 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. [152] on Tensorflow bug-related artefacts (from StackOverflow QA page and Github) found that nine out of 175 ML bugs (5.1%) belong to efficiency problems. This proportion is not high. The reasons may either be that efficiency problems rarely occur or 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 [189] 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 the 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 [190], 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 human beings.
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, it is less likely to model the minority group well.
5)*Proxies*: some features are proxies of sensitive attributes(e.g., neighbourhood), 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. Measuring and discovering the differences are actually defining and discovering fairness bugs. Such bugs can offend and even harm users, and cause programmers and businesses embarrassment, mistrust, loss of revenue, and even legal violations [161].

6.5.1 Fairness Definitions and Measurement Metrics

There are several definitions of fairness that proposed in the literature but with no firm consensus being yet reached [191], [192], [193], [194]. Even though, 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 that we are testing. 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 [195]. It is a relatively low-cost

way to define and ensure fairness. Nevertheless, sometimes the non-sensitive attributes in X may contain sensitive information correlated to those sensitive attributes that may still lead to discrimination [191], [195]. Excluding sensitive attributes may also impact model accuracy and yield less effective predictive results [196].

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 [197]. 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 has group fairness if $P\{h(x_i) = 1 | x_i \in G_1\} - P\{h(x_j) = 1 | x_j \in G_2\} < \epsilon$.

Equalised Odds is another group fairness approach proposed by Hardt et al. [132]. 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*** [132]. 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. [195] introduced ***Counter-factual Fairness***. A model h satisfies Counter-factual Fairness if its output remains the same when the protected attribute is flipped to a counter-factual value, and other variables modified as determined by the assumed causal model: $P\{h(x_i)_a | a \in A x_i \in X\} = P\{h(x_i)_{a'} = y_i | a \in A, x_i \in X\}$. This measurement of fairness additionally provides a mechanism to interpret the causes of bias.

4) Individual Fairness. Dwork et al. [131] 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: $P\{h(x_i) | x_i \in X\} = P\{h(x_j) = y_i | x_j \in X\}$ iff $d(x_i, x_j) < \epsilon$, where d is a distance metric for individuals that measures their similarity.

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 [191] surveyed how fairness is defined and formalised in the literature. Corbett-Davies and Goel [61] 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 [192] explained and illustrated the existing most prominent fairness definitions based on a common, unifying dataset.

Saxena et al. [193] 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.

6.5.2 Test Generation Techniques for Fairness Testing

Galhotra et al. [5], [198] proposed Themis which considers group fairness using causal analysis [199]. 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. [94] 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. Except for detecting fairness bugs, *Aeqitas* also retrains the machine-learning models and reduce discrimination in the decisions made by these models.

Agarwal et al. [102] 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[8] to identify whether factors that drive decision 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. [161] firstly 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 [116] tried to figure out where the unfairness comes from 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 shuffle the 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 human beings in the loop. That is, manual

[8] Local Interpretable Model-agnostic Explanations produces decision trees corresponding to an input that could provide paths in symbolic execution [103]

assessment is currently the major approach to evaluate interpretability. Doshi-Velez and Kim [64] gave a taxonomy of evaluation (testing) approaches for interpretability: application-grounded, human-grounded, and functionally-grounded. Application-grounded evaluation involves human beings in the experiments with a real application scenario. Human-grounded evaluation is to do the evaluation with real humans but with 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. [200] 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 1000 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 learned has actually learned the object in object identification scenario via occluding the surroundings of the objects.

Christoph [69] proposed to measure the interpretability based on the category of ML algorithms. He mentioned 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. [201] defined the concepts of metamorphic relation patterns (MRPs) and metamorphic relation input patterns (MRIPs) that can be adopted to help end users understand how an ML system really 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. [202] investigated several interpretability improving methods which transform classifier scores to the probability of disease scale. They showed that classifier scores on arbitrary scales can be calibrated to the probability scale without affecting their discrimination performance.

7 ML Testing Components

This section organises the related work of ML testing based on which ML component (data, learning program, or framework) ML testing tests.

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 more data, constructing a feedback loop that may amplify even small data bugs over time.

Nevertheless, data testing is challenging [203]. 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; what's more, the resilience of some ML algorithms to noisy data and the problems in correctness metrics add more difficulty to observe the problems in data.

7.1.1 Bug Detection in Training Data

Rule-based Data Bug Detection. Hynes et al. [168] proposed *data linter*– a ML tool inspired by code linters to automatically inspect ML datasets. They considered three types of data problems: 1) miscoding 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. [154] 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 % 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. [204] proposed to augment DNNs with a small sub-network specially designed to distinguish genuine data from data containing adversarial perturbations. Wang et al. [205] used DNN model mutation to expose adversarial examples since they found that adversarial samples are more sensitive to perturbations [206]. The evaluation was based on the MNIST and CIFAR10 datasets. The approach detects 96.4 %/90.6 % adversarial samples with 74.1/86.1 mutations for MNIST/CIFAR10.

Adversarial example detection can be regarded as bug detection in the test data. Carlini and Wagner [207] surveyed ten proposals that are designed for detecting adversarial examples and compared their efficacy. They found that detection approaches rely on the 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. We introduced the approaches in detecting test

data insufficiency in Section 5.3, such as coverage [1], [72], [85] and mutation score [145].

7.1.3 Skew Detection in Training and Test Data

The training instances and the instances that the model predicts should be consistent in aspects such as features and distributions. Kim et al. [121] proposed two measurements to evaluate the skew between training and test data: 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).

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 may not work because produce teams have difficulty in understanding the natural meaning of these metrics. Instead, they proposed to use the largest change in probability for 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 shows 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. [208], [209] 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. [169] 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 [210] to improve a model's performance. After resolving the problems detected, the tool is able to improve prediction accuracy by up to 9% in comparison to the best non-ensemble alternative.

ActiveClean and BoostClean may involve human beings in the loop of testing process. Schelter et al. [170] 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 [211] 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. [171]. 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 [212].

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 [213] introduced ML unit testing with TensorFlow built-in testing functions to help ensure that 'code will function as expected' to help build developers'.

Schaul et al. [214] 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. [215] and Guo et al. [216] identified operating systems, language, and hardware Compatibility issues. Sun et al. [215] studied 329 real bugs from three machine learning frameworks: Scikit-learn, Paddle, and Caffe. Over 22% bugs are found to be compatibility problems due to incompatible operating systems, language versions, or conflicts with hardware. Guo et al. [216] 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. [152] 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% (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. [217] 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 still remains unevaluated.

Algorithm Selection Examination. Developers usually face more than one learning algorithms to choose from. Fu and Menzies [218] 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. [219] 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. [111], [122] used mutants to simulate programming code errors to investigate whether the proposed metamorphic relations are effective in 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. [220] 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 and detecting framework relevant bugs.

7.3.1 Study of Framework Bugs

Xiao et al. [221] focused on the security vulnerabilities of popular deep learning frameworks including Caffe, TensorFlow, and Torch. They examined the code of popular deep learning frameworks of these frameworks. The dependency of these frameworks is found to be very complex. Multiple vulnerabilities are identified to exist in the implementation of these frameworks. The most common vulnerabilities are software bugs that cause programs to crash, or enter an infinite loop, or exhaust all the memory.

Guo et al. [216] 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 quite 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. [215] found that approximately 10% of the 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 Framework Implementation Testing

Many learning algorithms are implemented inside ML frameworks. Implementation bugs in ML frameworks may cause neither crashes, errors, nor efficiency problems [222], making their detection challenging.

Challenges in Detecting Implementation Bugs. Thung et al. [151] studied machine learning bugs as early as in 2012. The results regarding 500 bug reports from three machine learning systems indicated that approximately 22.6% bugs are due to incorrect implementation of defined algorithms. Cheng et al. [223] injected implementation bugs into classic machine learning code in Weka and observed the performance changes that result. They found that 8% to 40% of the logically non-equivalent executable mutants (injected implementation bugs) were statistically indistinguishable from their original versions.

Solutions towards Detecting Implementation Bugs. Some work has used multiple implementations or differential testing to detect bugs. For example, Alebiosu et al. [128] 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 (more details could be referred to Section 5.2.2).

However, not every algorithm has multiple implementations, at which case metamorphic testing may be helpful. Murphy et al. [10], [109] 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 not affect the 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, on three machine learning applications, found metamorphic testing to be an efficient and effective approach to testing ML applications.

Xie et al. [112] 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 [224]. Among the 43 injected faults in Weka [113] (injected by MuJava [225]), the metamorphic relations were able to reveal 39, indicating effectiveness. In their work, the test inputs were randomly generated data. More evaluation with real-world data is needed.

Dwarakanath et al. [115] 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, linear scaling of the test features. For deep learning models such as residual networks, since 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 to simulate implementation bugs, of which the proposed MRs are able to find 71%.

8 Application Scenarios

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

8.1 Autonomous Driving

Testing autonomous vehicles has a long history. Wegener and Bühler discussed compared different fitness functions when evaluating the tests of autonomous car parking systems [226].

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 in the case of failure [153], as was the case with the work of Wegener and Bühler [226]. A failure that causes the human driver to take control of the vehicle is called a *disengagement*.

Banerjee et al. [153] 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%) of which led to accidents. They classified the causes of disengagements into 10 types. As high as 64% of the disengagements were found to be caused by the bugs in the machine learning system, among which the low performance of image classification (e.g., improper detection of traffic lights, lane markings, holes, and bumps) were the dominant causes accounting for 44% of all reported disengagements. The remaining 20% 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. [72] proposed to use a set of image transformation to generate tests, which simulate the potential noises that could happen to a real-world camera. Zhang et al. [75] 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. [77] 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. [86] used feature-guided Monte Carlo Tree Search to identify elements of an image which are most vulnerable to a self-driving system to generate adversarial examples. Jha et al. [89] accelerated the process of finding 'safety-critical' 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. [87] 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. [160] 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[9].

Tuncali et al. [95] 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. [88] 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 [187] and Kayotee [188], which are fault injection-based tools to systematically inject faults into the autonomous driving systems to assess their safety and reliability.

8.2 Machine Translation

Machine translation automatically translates text or speech from one language to another. The BLEU (BiLingual Evaluation Understudy) score [227] 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. [123], [124] used metamorphic relations in their tool 'MT4MT' to evaluate 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.

The work of Zheng et al. [228], [229], [230] 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.

8.3 Natural Language Inference

A Nature Language Inference (NLI) 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. [91] generated sentence mutants (called 'rule-based adversaries' in the paper) to test whether the existing NLI models have semantic understanding. Surprisingly, 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.

[9] https://github.com/shromonag/FalsifyNN

Similarly, Wang et al. [92] mutated the inference target pair by simply swapping them. The heuristic is that a good NLI model should report comparable accuracy between the original test set and swapped test set for contradiction pairs and neutral pairs, and 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 summarise the datasets (name, description, size, and usage scenario of each dataset) that have been used in ML testing.

9.1 Timeline

Figure 8 shows several big events 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. Their testing mainly refers to 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. [109] discussed the properties of machine learning algorithms that may be adopted as metamorphic relations to detect implementation bugs.

In 2009, Xie et al. [112] also applied metamorphic testing on supervised learning applications.

The testing problem of fairness was proposed in 2012 by Dwork et al. [131]; the problem of interpretability was proposed in 2016 by Burrell [231].

In 2017, Pei et al. [1] published the first white-box testing paper on deep learning systems. This has become the milestone of machine learning testing, which pioneered to propose coverage criteria for DNN. Enlightened by this paper, a number of machine learning testing techniques have emerged, such as DeepTest [72], DeepGauge [85], DeepConcolic [104], and DeepRoad [75]. A number of software testing techniques has been applied to ML testing, such as different testing coverage criteria [72], [85], [138], mutation testing [145], combinatorial testing [142], metamorphic testing [93], and fuzz testing [82].

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 the research trend of ML testing, we classify the papers into two categories: those targeting only deep learning and those designed for general machine learning (including deep learning).

Among all the 128 papers, 52 papers (40.6%) present testing techniques that are specially designed for deep learning; the remaining 76 papers cater to 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. Figure 9 shows the results. Before 2017, papers mostly focus on general machine learning; after 2018, both general machine learning learning and deep learning testing notably arise.

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. Perhaps a bit surprisingly, almost all the work we identified in this survey focused on testing supervised machine learning. Among the 128 related papers, there are currently only three papers testing unsupervised machine learning: Murphy et al. [109] 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 k-means algorithm, which is a clustering algorithm. Xie et al. [119] designed metamorphic relations for unsupervised learning. One paper focused on reinforcement learning testing: Uesato et al. [87] proposed a predictive adversarial example generation approach to predict failures and estimate reliable risks in reinforcement learning.

We then tried to understand whether the imbalanced testing distribution among different categories is due to the imbalance of their research popularity. 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, 115 out of 128 papers were observed for supervised learning testing purpose. The table shows that testing popularity of different categories is obviously disproportionate 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 has different tasks such as classification, regression, clustering, and dimension reduction (see more in Section 2). The research focus on different tasks also shows to be highly

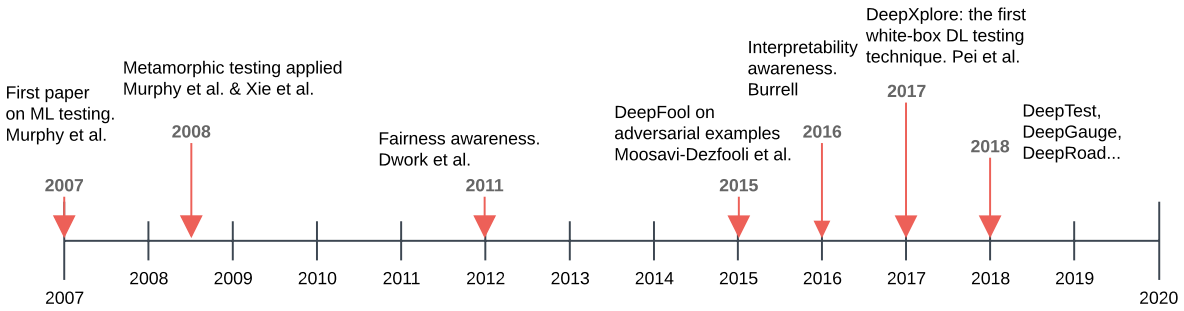

Figure 8: Timeline of ML testing research

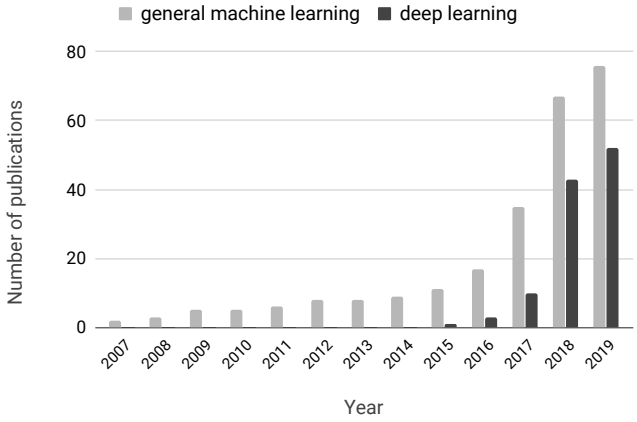

Figure 9: Commutative trends in general machine learning and deep learning

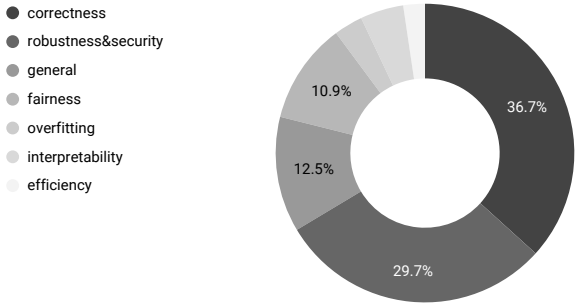

Figure 10: Research distribution among different testing properties

Table 4: Search hits and testing distribution of supervised/unsupervised/reinforcement Learning

Category	Scholar hits	Google hits	Testing hits
Supervised	1,610k	73,500k	115/128
Unsupervised	619k	17,700k	3/128
Reinforcement	2,560k	74,800k	1/128

imbalanced: among the papers we identified, almost all of them focus on classification.

9.3 Research Distribution among Different Testing Properties

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

From the figure, around one-third (32.1%) of the papers test correctness. Another one-third of the papers focus on robustness and security problems. Fairness testing ranks the third among all the properties, with 13.8% papers.

Nevertheless, for overfitting detection, interpretability testing, and efficiency testing, only three papers exist for each in our paper collection. For privacy, there are some papers discussing how to ensure privacy [232], but we did not find papers that systematically 'test' privacy or detect privacy violations.

9.4 Datasets Used in ML Testing

Tables 5 to 8 show the details for some widely-adopted datasets used in ML testing research. In each table, the first column shows the name and link of each dataset. The next a few columns give a brief description, the size (training set size + test set size if applicable), the testing problem(s), the usage application scenario of each dataset[10].

Table 5 shows the datasets used for image classification tasks. Most datasets are often very 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 the correctness, overfitting, and robustness of ML systems.

Table 6 shows the datasets related to natural language processing. The contents are usually texts, sentences, or files with texts, with the usage 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 the evaluation (for those papers with an evaluation). Figure 11 shows the results. Surprisingly, most papers use only one or two datasets in the evalu-

[10] These tables do not list data cleaning datasets because many data can be dirty and some evaluation may involve hundreds of data sets [169]

Table 5: Datasets (1/4): Image Classification

Dataset	Description	Size	Usage
MNIST [233]	Images of handwritten digits	60,000+10,000	correctness, overfitting, robustness
Fashion MNIST [234]	MNIST-like dataset of fashion images	70,000	correctness, overfitting
CIFAR-10 [235]	General images with 10 classes	50,000+10,000	correctness, overfitting, robustness
ImageNet [236]	Visual recognition challenge dataset	14,197,122	correctness, robustness
IRIS flower [237]	The Iris flowers	150	overfitting
SVHN [238]	House numbers	73,257+26,032	correctness,robustness
Fruits 360 [239]	Dataset with 65,429 images of 95 fruits	65,429	correctness,robustness
Handwritten Letters [240]	Colour images of Russian letters	1,650	correctness,robustness
Balance Scale [241]	Psychological experimental results	625	overfitting
DSRC [242]	Wireless communications between vehicles and road side units	10,000	overfitting, robustness
Udacity challenge [73]	Udacity Self-Driving Car Challenge images	101,396+5,614	robustness
Nexar traffic light challenge [243]	Dashboard camera	18,659+500,000	robustness
MSCOCO [244]	Object recognition	160,000	correctness
Autopilot-TensorFlow [245]	Recorded to test the NVIDIA Dave model	45,568	robustness
KITTI [246]	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 [247]	questions and answers for NLP	1000+1000	robustness
Tiny Shakespeare [248]	Samples from actual Shakespeare	100,000 character	correctness
Stack Exchange Data Dump [249]	Stack Overflow questions and answers	365 files	correctness
SNLI [250]	Stanford Natural Language Inference Corpus	570,000	robustness
MultiNLI [251]	Crowd-sourced collection of sentence pairs annotated with textual entailment information	433,000	robustness
DMV failure reports [252]	AV failure reports from 12 manufacturers in California[11]	keep updating	correctness

ation; 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 [168].

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 [198], an open-source tool for testing group discrimination[12]. There is also an ML testing framework for tensorflow, named *mltest*[13], 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*[14]. Dolby et al. [220] extended WALA to enable static analysis for machine learning code using TensorFlow.

Overall, unlike traditional testing, the existing tool support in ML testing is immature. There is still enormous 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).

[12] http://fairness.cs.umass.edu/

[13] https://github.com/Thenerdstation/mltest

[14] https://github.com/suriyadeepan/torchtest

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

Dataset	Description	Size	Usage
German Credit [253]	Descriptions of customers with good and bad credit risks	1,000	fairness
Adult [254]	Census income	48,842	fairness
Bank Marketing [255]	Bank client subscription term deposit data	45,211	fairness
US Executions [256]	Records of every execution performed in the United States	1,437	fairness
Fraud Detection [257]	European Credit cards transactions	284,807	fairness
Berkeley Admissions Data [258]	Graduate school applicants to the six largest departments at University of California, Berkeley in 1973	4,526	fairness
Broward County COMPAS [259]	Score to determine whether to release a defendant	18,610	fairness
MovieLens Datasets [260]	People's preferences for movies	100k-20m	fairness
Zestimate [261]	data about homes and Zillow's in-house price and predictions	2,990,000	correctness
FICO scores [262]	United States credit worthiness	301,536	fairness
Law school success [263]	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 [264]	Malicious PDF files	5,000	robustness
Contagio [265]	Clean and malicious files	28,760	robustness
Drebin [266]	Applications from different malware families	123,453	robustness
Chess [267]	Chess game data: King+Rook versus King+Pawn on a7	3,196	correctness
Waveform [241]	CART book's generated waveform data	5,000	correctness

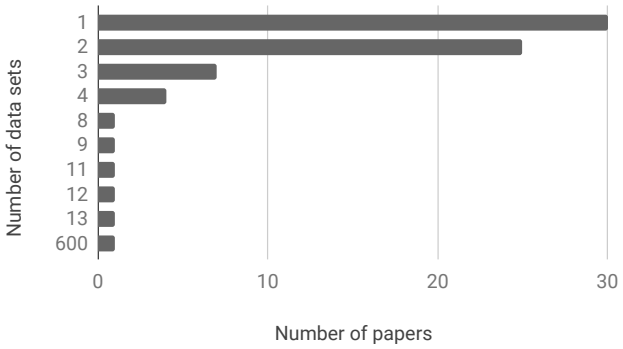

Figure 11: Number of papers with different amounts of datasets in experiments

10.1 Challenges in ML Testing

ML testing has experienced rapid progress. Nevertheless, ML testing is still at an early stage, with many challenges and open questions ahead.

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

Search-based test generation (SBST) [80] uses a meta-heuristic optimising search technique, such as a Genetic Algorithm, to automatically generate test inputs. It is one of the key automatic test generation techniques in traditional software interesting. Except for generating test inputs for testing functional properties like program correctness, SBST has also been used to explore tensions in algorithmic fairness in requirement analysis. [194], [268]. There exist huge research opportunities of applying SBST on generating test inputs for testing ML systems, since SBST has the advantage of searching test inputs in very large input space.

The existing test input generation techniques focus more on generating adversarial input to test the robustness of an ML system. However, adversarial examples are often arguably criticised that they do not represent the real input data that could happen in practice. Thus, an interesting research direction is to how to generate natural test inputs and how to automatically measure the naturalness of the generated inputs.

We have seen some related work that tries to generate test inputs as natural as possible under the scenario of autonomous driving, such as DeepTest [72], DeepHunter [85] and DeepRoad [75], yet the generated images could still suffer from unnaturalness: sometimes even human beings could not recognise the generated images by these tools. It is both interesting and challenging to explore

whether such kinds of test data that are meaningless to human beings should be adopted/valid in ML testing.

Challenges on Test Assessment Criteria. There have 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 correlated with each other, or how these assessment metrics correlate with the fault-revealing ability of tests, which has been maturaly studied in traditional software testing [269].

Challenges in The Oracle Problem. Oracle problem remains a challenge in ML testing. Metamorphic relations are effective pseudo oracles, but are proposed by human beings in most cases, and may contain false positives. A big challenge is thus to automatically identify and construct reliable test oracles for ML testing.

Murphy et al. [122] discussed that flaky tests are likely to come up in metamorphic testing whenever floating point calculations are involved. Flaky test detection is a very challenging problem in traditional software testing [270]. It is even more challenging in ML testing because of the oracle problem.

Even without flaky tests, pseudo oracles may sometimes be inaccurate, leading to many false positives. There is a need to explore how to yield more accurate test oracles or 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 is 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 even more serious, especially when testing the ML component, because ML component testing usually needs retraining of the model or repeating of the prediction process, as well as data generation to explore the enormous mode behaviour space.

A possible research direction of reducing cost is to represent an ML model into some kind of intermediate state to make it easier for testing.

We could also apply traditional cost reduction techniques such as test prioritisation or test minimisation to reduce the size of test cases while remaining the test correctness.

As more and 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 remains many research opportunities in ML testing.

Testing More Application Scenarios. Many current research focuses on supervised learning, in particular the classification problem. More research works are highly desired for unsupervised learning and reinforcement learning.

The testing task mostly centres around image classification. There are also opportunities in many other areas such as speech recognition, natural language processing.

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

Transfer learning is gaining popularity recently, which focuses on storing knowledge gained while solving one problem and applying it to a different but related problem [271]. Therefore, transfer learning testing is also important.

Testing More Properties. From Figure 10, most work test robustness and correctness, while only less than 3% papers study efficiency, overfitting detection, or interpretability. We did not find any papers that systematically test data privacy violations.

In particular, for testing property interpretability, the existing approaches still mainly rely on manual assessment, which check whether human beings could understand the logic or predictive results of an ML model. It is interesting to investigate the automatic assessment of interpretability or detection of interpretability violations.

There has been a discussion that machine learning testing and traditional software testing may have different requirements in the assurance level towards different properties [272]. It might also be interesting to explore what properties are the most important for machine learning systems, and thus deserve more research and testing efforts.

Presenting More Testing Benchmarks A huge number of datasets have been adopted in the existing ML testing papers. Nevertheless, as Tables 5 to 8 show, the datasets are usually those adopted in building machine learning systems. As far as we know, there are very few benchmarks like CleverHans[15] that are specially designed for the ML testing research (i.e., adversarial example construction) purpose.

We hope that in the future, more benchmarks that are specially designed for ML testing could be presented For example, a repository of machine learning programs with real bugs could present a good benchmark for bug-fixing techniques, like Defects4J[16] in traditional software testing.

Testing More Types of Testing Activities. From the introduction in Section 5, as far we know, the requirement analysis of ML systems is still absent in ML testing. Demonstrated by Finkelstein et al. [194], [268], a good requirement analysis may tackle many non-functional properties such as fairness.

The existing work is mostly about off-line testing. Online-testing deserves many research efforts as well. Nevertheless, researchers from academia may have limitations in assessing online models or data, it might be necessary to cooperate with industry to conduct online testing.

According to the work of Amershi et al. [8], data testing is especially important and certainly deserves more research efforts on it. Additionally, there are also many opportunities 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 much more difficult, sometimes even impossible, for developers to understand than in traditional software testing. Visualisation of testing

[15] https://github.com/tensorflow/cleverhans
[16] https://github.com/rjust/defects4j

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 [122], [223], 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, which we believe could be 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 workflow. It also summarised the datasets used for experiments and the available open-source testing tools/frameworks, and analysed the research trend, directions, opportunities, and challenges in ML testing. We hope this survey could help software engineering and machine learning researchers get familiar with the literature research status of ML testing quickly and thoroughly, and orientate more researchers to contribute to the pressing problems of ML testing.

Acknowledgement

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 an earlier draft of this paper.

References

[1] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[2] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[3] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[4] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[5] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 498–510. ACM, 2017.

[6] Maciej Pacula. Unit-Testing Statistical Software. http://blog.mpacula.com/2011/02/17/unit-testing-statistical-software/, 2011.

[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 *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 786–791, March 2016.

[8] Saleema Amershi, Andrew Begel, Christian Bird, Rob DeLine, Harald Gall, Ece Kamar, Nachi Nagappan, Besmira Nushi, and Tom Zimmermann. Software engineering for machine learning: A case study. In *Proc. ICSE (Industry Track, to appear)*, 2019.

[9] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[10] Chris Murphy, Gail E Kaiser, and Marta Arias. An approach to software testing of machine learning applications. In *SEKE*, volume 167, 2007.

[11] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM 81 Conference*, ACM 81, pages 254–257, 1981.

[12] Kelly Androutsopoulos, David Clark, Haitao Dan, Mark Harman, and Robert Hierons. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In 36^{th} *International Conference on Software Engineering (ICSE 2014)*, pages 573–583, Hyderabad, India, June 2014.

[13] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[14] David Clark and Robert M. Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8–9):335 – 340, 2012.

[15] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing (best paper award winner). In *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 2008. IEEE Computer Society.

[16] Christopher D Turner and David J Robson. The state-based testing of object-oriented programs. In *1993 Conference on Software Maintenance*, pages 302–310. IEEE, 1993.

[17] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[18] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[19] Atif M Memon. Gui testing: Pitfalls and process. *Computer*, (8):87–88, 2002.

[20] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 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 *2018 Annual IEEE International Systems Conference (SysCon)*, pages 1–5, April 2018.

[22] Lei Ma, Felix Juefei-Xu, Minhui Xue, Qiang Hu, Sen Chen, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Secure deep learning engineering: A software quality assurance perspective. *arXiv preprint arXiv:1810.04538*, 2018.

[23] Xiaowei Huang, Daniel Kroening, Marta Kwiatkowska, Wenjie Ruan, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. Safety and trustworthiness of deep neural networks: A survey. *arXiv preprint arXiv:1812.08342*, 2018.

[24] Satoshi Masuda, Kohichi Ono, Toshiaki Yasue, and Nobuhiro Hosokawa. A survey of software quality for machine learning applications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 279–284. IEEE, 2018.

[25] Fuyuki Ishikawa. Concepts in quality assessment for machine learning-from test data to arguments. In *International Conference on Conceptual Modeling*, pages 536–544. Springer, 2018.

[26] Houssem Braiek and Foutse Khomh. On testing machine learning programs. *arXiv preprint arXiv:1812.02257*, 2018.

[27] John Shawe-Taylor, Nello Cristianini, et al. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[28] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[30] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

[32] François Chollet et al. Keras. https://keras.io, 2015.

[33] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[34] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[35] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[36] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[37] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[38] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

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

[40] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[41] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, Jan 2010.

[42] Roman Werpachowski, András György, and Csaba Szepesvári. Detecting overfitting via adversarial examples. *arXiv preprint arXiv:1903.02380*, 2019.

[43] Juan Cruz-Benito, Andrea Vázquez-Ingelmo, José Carlos Sánchez-Prieto, Roberto Therón, Francisco José García-Peñalvo, and Martín Martín-González. Enabling adaptability in web forms based on user characteristics detection through a/b testing and machine learning. *IEEE Access*, 6:2251–2265, 2018.

[44] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *The Journal of Machine Learning Research*, 17(1):1–42, 2016.

[45] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data Validation for Machine Learning. In *SysML*, 2019.

[46] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Hirotoshi Yasuoka. Towards dependability metrics for neural networks. *arXiv preprint arXiv:1806.02338*, 2018.

[47] Scott Alfeld, Xiaojin Zhu, and Paul Barford. Data poisoning attacks against autoregressive models. In *AAAI*, pages 1452–1458, 2016.

[48] Weizhen Qi Lin Tan Viet Hung Pham, Thibaud Lutellier. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *Proc. ICSE*, 2019.

[49] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[50] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[51] Matthew Kirk. *Thoughtful machine learning: A test-driven approach*. " O'Reilly Media, Inc.", 2014.

[52] Vladimir Vapnik, Esther Levin, and Yann Le Cun. Measuring the vc-dimension of a learning machine. *Neural computation*, 6(5):851–876, 1994.

[53] David S Rosenberg and Peter L Bartlett. The rademacher complexity of co-regularized kernel classes. In *Artificial Intelligence and Statistics*, pages 396–403, 2007.

[54] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[55] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1–17, 2013.

[56] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[57] Cynthia Dwork. Differential privacy. *Encyclopedia of Cryptography and Security*, pages 338–340, 2011.

[58] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[59] wikipedia. California Consumer Privacy Act. https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act.

[60] R. Baeza-Yates and Z. Liaghat. Quality-efficiency trade-offs in machine learning for text processing. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 897–904, Dec 2017.

[61] Sam Corbett-Davies and Sharad Goel. The measure and mismeasure of fairness: A critical review of fair machine learning. *arXiv preprint arXiv:1808.00023*, 2018.

[62] Drew S Days III. Feedback loop: The civil rights act of 1964 and its progeny. *. Louis ULJ*, 49:981, 2004.

[63] Zachary C Lipton. The mythos of model interpretability. *arXiv preprint arXiv:1606.03490*, 2016.

[64] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[65] Thibault Sellam, Kevin Lin, Ian Yiran Huang, Michelle Yang, Carl Vondrick, and Eugene Wu. Deepbase: Deep inspection of neural networks. In *Proc. SIGMOD*, 2019.

[66] Or Biran and Courtenay Cotton. Explanation and justification in machine learning: A survey. In *IJCAI-17 Workshop on Explainable AI (XAI)*, page 8, 2017.

[67] Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence*, 2018.

[68] Bryce Goodman and Seth Flaxman. European union regulations on algorithmic decision-making and a" right to explanation". *arXiv preprint arXiv:1606.08813*, 2016.

[69] Christoph Molnar. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/, 2019. https://christophm.github.io/interpretable-ml-book/.

[70] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 701–712. ACM, 2014.

[71] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 38. Citeseer, 2014.

[72] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, pages 303–314. ACM, 2018.

[73] udacity. Udacity challenge. https://github.com/udacity/self-driving-car.

[74] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[75] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 132–142, 2018.

[76] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems*, pages 700–708, 2017.

[77] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. *arXiv preprint arXiv:1812.10812*, 2018.

[78] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. Deepcruiser: Automated guided testing for stateful deep learning systems. *arXiv preprint arXiv:1812.05339*, 2018.

[79] Junhua Ding, Dongmei Zhang, and Xin-Hua Hu. A framework for ensuring the quality of a big data service. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 82–89. IEEE, 2016.

[80] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[81] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105. ACM, 2007.

[82] Augustus Odena and Ian Goodfellow. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. *arXiv preprint arXiv:1807.10875*, 2018.

[83] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. Dlfuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 739–743. ACM, 2018.

[84] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266*, 2018.

[85] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 120–131, 2018.

[86] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 408–426, 2018.

[87] Jonathan Uesato*, Ananya Kumar*, Csaba Szepesvari*, Tom Erez, Avraham Ruderman, Keith Anderson, Krishnamurthy (Dj) Dvijotham, Nicolas Heess, and Pushmeet Kohli. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. In *International Conference on Learning Representations*, 2019.

[88] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, 2019.

[89] Saurabh Jha, Subho S Banerjee, Timothy Tsai, Siva KS Hari, Michael B Sullivan, Zbigniew T Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Ml-based fault injection for autonomous vehicles.

[90] Sakshi Udeshi and Sudipta Chattopadhyay. Grammar based directed testing of machine learning systems. *arXiv preprint arXiv:1902.10027*, 2019.

[91] Yixin Nie, Yicheng Wang, and Mohit Bansal. Analyzing compositionality-sensitivity of nli models. *arXiv preprint arXiv:1811.07033*, 2018.

[92] Haohan Wang, Da Sun, and Eric 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. *arXiv preprint arXiv:1809.02719*, 2018.

[93] Alvin Chan, Lei Ma, Felix Juefei-Xu, Xiaofei Xie, Yang Liu, and Yew Soon Ong. Metamorphic relation based adversarial attacks on differentiable neural computer. *arXiv preprint arXiv:1809.02444*, 2018.

[94] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 98–108. ACM, 2018.

[95] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *IEEE Intelligent Vehicles Symposium (IV)*, 2018.

[96] Alan Hartman. Software and hardware testing using combinatorial covering suites. In *Graph theory, combinatorics and algorithms*, pages 237–266. Springer, 2005.

[97] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[98] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[99] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[100] Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758–1773, 2013.

[101] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S Pasareanu, and Sarfraz Khurshid. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*, 2018.

[102] Aniya Agarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. Automated test generation to detect individual discrimination in ai models. *arXiv preprint arXiv:1809.03260*, 2018.

[103] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[104] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 109–119, 2018.

[105] Christian Murphy, Gail Kaiser, and Marta Arias. Parameterizing Random Test Data According to Equivalence Classes. In *Proc. ASE*, RT '07, pages 38–41, 2007.

[106] Jie Zhang, Earl T Barr, Benjamin Guedj, Mark Harman, and John Shawe-Taylor. Perturbed Model Validation: A New Framework to Validate Model Relevance. working paper or preprint, May 2019.

[107] Shin Nakajima and Hai Ngoc Bui. Dataset coverage for testing machine learning computer programs. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 297–304. IEEE, 2016.

[108] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[109] Christian Murphy, Gail E. Kaiser, Lifeng Hu, and Leon Wu. Properties of machine learning applications for use in metamorphic testing. In *SEKE*, 2008.

[110] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. Validating a deep learning framework by metamorphic testing. In *Proceedings of the 2Nd International Workshop on Metamorphic Testing*, MET '17, pages 28–34, 2017.

[111] Christian Murphy, Kuang Shen, and Gail Kaiser. Using JML Runtime Assertion Checking to Automate Metamorphic Testing in Applications Without Test Oracles. In *Proc. ICST*, pages 436–445. IEEE Computer Society, 2009.

[112] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 135–144. IEEE, 2009.

[113] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[114] Shin Nakajima. Generalized oracle for testing machine learning computer programs. In *International Conference on Software Engineering and Formal Methods*, pages 174–179. Springer, 2017.

[115] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 118–128, 2018.

[116] Arnab Sharma and Heike Wehrheim. Testing machine learning algorithms for balanced data usage. In *Proc. ICST*, pages 125–135, 2019.

[117] Sadam Al-Azani and Jameleddine Hassine. Validation of machine learning classifiers using metamorphic testing and feature selection techniques. In *International Workshop on Multi-disciplinary Trends in Artificial Intelligence*, pages 77–91. Springer, 2017.

[118] Manikandasriram Srinivasan Ramanagopal, Cyrus Anderson, Ram Vasudevan, and Matthew Johnson-Roberson. Failing to learn: Autonomously identifying perception failures for self-driving cars. *IEEE Robotics and Automation Letters*, 3(4):3860–3867, 2018.

[119] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. Mettle: A metamorphic testing approach to validating unsupervised machine learning methods, 2018.

[120] Shin Nakajima. Dataset diversity for metamorphic testing of machine learning software. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 21–38. Springer, 2018.

[121] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. *arXiv preprint arXiv:1808.08444*, 2018.

[122] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic System Testing of Programs Without Test Oracles. In *18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 189–200, 2009.

[123] L. Sun and Z. Q. Zhou. Metamorphic Testing for Machine Translations: MT4MT. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 96–100, Nov 2018.

[124] Daniel Pesu, Zhi Quan Zhou, Jingfeng Zhen, and Dave Towey. A monte carlo method for metamorphic testing of machine translation services. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, pages 38–45. ACM, 2018.

[125] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[126] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Notices*, volume 49, pages 216–226. ACM, 2014.

[127] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[128] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. Multiple-implementation testing of supervised learning software. In *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.

[129] Yi Qin, Huiyan Wang, Chang Xu, Xiaoxing Ma, and Jian Lu. Syneva: Evaluating ml programs by mirror program synthesis. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 171–182. IEEE, 2018.

[130] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[131] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pages 214–226. ACM, 2012.

[132] Moritz Hardt, Eric Price, Nati Srebro, et al. Equality of opportunity in supervised learning. In *Advances in neural information processing systems*, pages 3315–3323, 2016.

[133] Indre Zliobaite. Fairness-aware machine learning: a perspective. *arXiv preprint arXiv:1708.00754*, 2017.

[134] Bernease Herman. The promise and peril of human evaluation for model interpretability. *arXiv preprint arXiv:1711.07414*, 2017.

[135] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. Model assertions for debugging machine learning. In *NeurIPS MLSys Workshop*, 2018.

[136] Lianghao Li and Qiang Yang. Lifelong machine learning test. In *Proceedings of the Workshop on âĂIJBeyond the Turing TestâĂİ of AAAI Conference on Artificial Intelligence*, 2015.

[137] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analaysis*, ISSTA 2016, pages 342–353, 2016.

[138] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. Testing deep neural networks. *arXiv preprint arXiv:1803.04792*, 2018.

[139] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the mc/dc coverage criterion on the hete-2 satellite software. In *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1B6–1. IEEE, 2000.

[140] Jasmine Sekhon and Cody Fleming. Towards improved testing for deep learning. In *Proc. ICSE(NIER track, to appear)*, 2019.

[141] Lei Ma, Fuyuan Zhang, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Combinatorial testing for deep learning systems. *arXiv preprint arXiv:1806.07723*, 2018.

[142] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. Deepct: Tomographic combinatorial testing for deep learning systems. In *Proc. SANER*, pages 614–618, 02 2019.

[143] Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. Structural coverage criteria for neural networks could be misleading. In *Proc. ICSE(NIER track, to appear)*, 2019.

[144] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[145] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepMutation: Mutation Testing of Deep Learning Systems, 2018.

[146] Weijun Shen, Jun Wan, and Zhenyu Chen. MuNN: Mutation Analysis of Neural Networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 108–115. IEEE, 2018.

[147] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 1123–1132. IEEE, 2017.

[148] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. Input prioritization for testing neural networks. *arXiv preprint arXiv:1901.03768*, 2019.

[149] Long Zhang, Xuechao Sun, Yong Li, Zhenyu Zhang, and Yang Feng. A noise-sensitivity-analysis-based test prioritization technique for deep neural networks. *arXiv preprint arXiv:1901.00054*, 2019.

[150] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lu. Boosting Operational DNN Testing Efficiency through Conditioning. In *Proc. FSE*, page to appear, 2019.

[151] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 271–280. IEEE, 2012.

[152] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.

[153] 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 *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 586–597, June 2018.

[154] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. pages 175–186, 2018.

[155] Shanqing Cai, Eric Breck, Eric Nielsen, Michael Salib, and D Sculley. Tensorflow debugger: Debugging dataflow graphs for machine learning. In *Proceedings of the Reliable Machine Learning in the Wild-NIPS 2016 Workshop*, 2016.

[156] Manasi Vartak, Joana M F da Trindade, Samuel Madden, and Matei Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1285–1300. ACM, 2018.

[157] Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, page 4. ACM, 2017.

[158] Besmira Nushi, Ece Kamar, Eric Horvitz, and Donald Kossmann. On human intellect and machine failures: Troubleshooting integrative machine learning systems. In *AAAI*, pages 1017–1025, 2017.

[159] Wei Yang and Tao Xie. Telemade: A testing framework for learning-based malware detection systems. In *AAAI Workshops*, 2018.

[160] Tommaso Dreossi, Shromona Ghosh, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. Systematic testing of convolutional neural networks for autonomous driving. *arXiv preprint arXiv:1708.03309*, 2017.

[161] Florian Tramer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. Fairtest: Discovering unwarranted associations in data-driven applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 401–416. IEEE, 2017.

[162] Yasuharu Nishi, Satoshi Masuda, Hideto Ogawa, and Keiji Uetsuki. A test architecture for machine learning product. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 273–278. IEEE, 2018.

[163] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[164] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[165] Nathalie Japkowicz. Why question machine learning evaluation methods. In *AAAI Workshop on Evaluation Methods for Machine Learning*, pages 6–11, 2006.

[166] Weijie Chen, Brandon D Gallas, and Waleed A Yousef. Classifier variability: accounting for training and testing. *Pattern Recognition*, 45(7):2661–2671, 2012.

[167] Weijie Chen, Frank W Samuelson, Brandon D Gallas, Le Kang, Berkman Sahiner, and Nicholas Petrick. On the assessment of the added value of new predictive biomarkers. *BMC medical research methodology*, 13(1):98, 2013.

[168] Nick Hynes, D Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. 2017.

[169] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299*, 2017.

[170] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, 11(12):1781–1794, 2018.

[171] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

[172] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[173] Heang-Ping Chan, Berkman Sahiner, Robert F Wagner, and Nicholas Petrick. Classifier design for computer-aided diagnosis: Effects of finite sample size on the mean performance of classical and neural network classifiers. *Medical physics*, 26(12):2654–2668, 1999.

[174] Berkman Sahiner, Heang-Ping Chan, Nicholas Petrick, Robert F Wagner, and Lubomir Hadjiiski. Feature selection and classifier performance in computer-aided diagnosis: The effect of finite sample size. *Medical physics*, 27(7):1509–1522, 2000.

[175] Keinosuke Fukunaga and Raymond R. Hayes. Effects of sample size in classifier design. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (8):873–885, 1989.

[176] Alexej Gossmann, Aria Pezeshk, and Berkman Sahiner. Test data reuse for evaluation of adaptive machine learning algorithms: over-fitting to a fixed'test'dataset and a potential solution. In *Medical Imaging 2018: Image Perception, Observer Performance, and Technology Assessment*, volume 10577, page 105770K. International Society for Optics and Photonics, 2018.

[177] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[178] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in neural information processing systems*, pages 2613–2621, 2016.

[179] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[180] Wenjie Ruan, Min Wu, Youcheng Sun, Xiaowei Huang, Daniel Kroening, and Marta Kwiatkowska. Global robustness evaluation of deep neural networks with provable guarantees for l0 norm. *arXiv preprint arXiv:1804.05805*, 2018.

[181] Divya Gopinath, Guy Katz, Corina S Păsăreanu, and Clark Barrett. Deepsafe: A data-driven approach for assessing robustness of neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 3–19. Springer, 2018.

[182] Ravi Mangal, Aditya Nori, and Alessandro Orso. Robustness of neural networks: A probabilistic and practical perspective. In *Proc. ICSE(NIER track, to appear)*, 2019.

[183] Subho S Banerjee, James Cyriac, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Towards a bayesian approach for assessing faulttolerance of deep neural networks. In *Proc. DSN (extended abstract)*, 2019.

[184] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[185] Nicolas Papernot, Ian Goodfellow, Ryan Sheatsley, Reuben Feinman, and Patrick McDaniel. cleverhans v1.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 2016.

[186] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[187] Saurabh Jha, Subho S Banerjee, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. AVFI: Fault injection for autonomous vehicles. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 55–56. IEEE, 2018.

[188] Saurabh Jha, Timothy Tsai, Siva Hari, Michael Sullivan, Zbigniew Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors. In *3rd IEEE International Workshop on Automotive Reliability & Test*, 2018.

[189] Helge Spieker and Arnaud Gotlieb. Towards testing of deep learning systems with training set reduction. *arXiv preprint arXiv:1901.04169*, 2019.

[190] Solon Barocas and Andrew D Selbst. Big data's disparate impact. *Cal. L. Rev.*, 104:671, 2016.

[191] Pratik Gajane and Mykola Pechenizkiy. On formalizing fairness in prediction with machine learning. *arXiv preprint arXiv:1710.03184*, 2017.

[192] Sahil Verma and Julia Rubin. Fairness definitions explained. In *International Workshop on Software Fairness*, 2018.

[193] Nripsuta Saxena, Karen Huang, Evan DeFilippis, Goran Radanovic, David Parkes, and Yang Liu. How do fairness definitions fare? examining public attitudes towards algorithmic definitions of fairness. *arXiv preprint arXiv:1811.03654*, 2018.

[194] Anthony Finkelstein, Mark Harman, Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. Fairness analysis in requirements assignments. In 16^{th} *IEEE International Requirements Engineering Conference*, pages 115–124, Los Alamitos, California, USA, September 2008.

[195] Matt J Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. Counterfactual fairness. In *Advances in Neural Information Processing Systems*, pages 4066–4076, 2017.

[196] Nina Grgic-Hlaca, Muhammad Bilal Zafar, Krishna P Gummadi, and Adrian Weller. The case for process fairness in learning: Feature selection for fair decision making. In *NIPS Symposium on Machine Learning and the Law, Barcelona, Spain*, volume 8, 2016.

[197] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. Fairness constraints: Mechanisms for fair classification. *arXiv preprint arXiv:1507.05259*, 2015.

[198] Rico Angell, Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Themis: Automatically testing software for discrimination. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 871–875. ACM, 2018.

[199] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. Causal testing: Finding defects' root causes. *CoRR*, abs/1809.06991, 2018.

[200] Sorelle A. Friedler, Chitradeep Dutta Roy, Carlos Scheidegger, and Dylan Slack. Assessing the Local Interpretability of Machine Learning Models. *CoRR*, abs/1902.03501, 2019.

[201] Zhi Quan Zhou, Liqun Sun, Tsong Yueh Chen, and Dave Towey. Metamorphic relations for enhancing system understanding and use. *IEEE Transactions on Software Engineering*, 2018.

[202] Weijie Chen, Berkman Sahiner, Frank Samuelson, Aria Pezeshk, and Nicholas Petrick. Calibration of medical diagnostic classifier scores to the probability of disease. *Statistical methods in medical research*, 27(5):1394–1409, 2018.

[203] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1723–1726. ACM, 2017.

[204] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. *arXiv preprint arXiv:1702.04267*, 2017.

[205] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *Proc. ICSE (to appear)*, 2019.

[206] Jingyi Wang, Jun Sun, Peixin Zhang, and Xinyu Wang. Detecting adversarial samples for deep neural networks through mutation testing. *CoRR*, abs/1805.05010, 2018.

[207] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 3–14. ACM, 2017.

[208] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. Activeclean: interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment*, 9(12):948–959, 2016.

[209] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, Jiannan Wang, and Eugene Wu. Activeclean: An interactive data cleaning framework for modern machine learning. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2117–2120. ACM, 2016.

[210] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[211] Sanjay Krishnan and Eugene Wu. Alphaclean: automatic generation of data cleaning pipelines. *arXiv preprint arXiv:1904.11827*, 2019.

[212] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[213] Nick McClure. *TensorFlow machine learning cookbook*. Packt Publishing Ltd, 2017.

[214] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. In *ICLR 2014*, 2014.

[215] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li. An empirical study on real bugs for machine learning programs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 348–357, 2017.

[216] An Orchestrated Empirical Study on Deep Learning Frameworks and Lei Ma Qiang Hu Ruitao Feng Li Li Yang Liu Jianjun Zhao Xiaohong Li Platforms Qianyu Guo, Xiaofei Xie. An orchestrated empirical study on deep learning frameworks and platforms. *arXiv preprint arXiv:1811.05187*, 2018.

[217] Yu. L. Karpov, L. E. Karpov, and Yu. G. Smetanin. Adaptation of general concepts of software testing to neural networks. *Programming and Computer Software*, 44(5):324–334, Sep 2018.

[218] Wei Fu and Tim Menzies. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 49–60. ACM, 2017.

[219] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: How far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 373–384. ACM, 2018.

[220] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: Analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 1–10, 2018.

[221] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 123–128. IEEE, 2018.

[222] Chase Roberts. How to unit test machine learning code, 2017.

[223] Dawei Cheng, Chun Cao, Chang Xu, and Xiaoxing Ma. Manifesting bugs in machine learning code: An explorative study with mutation testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 313–324. IEEE, 2018.

[224] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.*, 84(4):544–558, April 2011.

[225] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[226] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference*, pages 1400–1412. Springer, 2004.

[227] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[228] Wujie Zheng, Wenyu Wang, Dian Liu, Changrong Zhang, Qinsong Zeng, Yuetang Deng, Wei Yang, Pinjia He, and Tao Xie. Testing untestable neural machine translation: An industrial case. 2018.

[229] Wujie Zheng, Wenyu Wang, Dian Liu, Changrong Zhang, Qinsong Zeng, Yuetang Deng, Wei Yang, Pinjia He, and Tao Xie. Testing untestable neural machine translation: An industrial case. In *ICSE (poster track)*, 2019.

[230] Wenyu Wang, Wujie Zheng, Dian Liu, Changrong Zhang, Qinsong Zeng, Yuetang Deng, Wei Yang, Pinjia He, and Tao Xie. Detecting failures of neural machine translation in the absence of reference translations. In *Proc. DSN (industry track)*, 2019.

[231] Jenna Burrell. How the machine âĂŸthinksâĂŹ: Understanding opacity in machine learning algorithms. *Big Data & Society*, 3(1):2053951715622512, 2016.

[232] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 399–414, April 2018.

[233] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[234] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[235] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[236] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[237] R.A. Fisher. Iris Data Set . http://archive.ics.uci.edu/ml/datasets/iris.

[238] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011.

[239] Horea MureÈŹan and Mihai Oltean. Fruit recognition from images using deep learning. *Acta Universitatis Sapientiae, Informatica*, 10:26–42, 06 2018.

[240] Olga Belitskaya. Handwritten Letters. https://www.kaggle.com/olgabelitskaya/handwritten-letters.

[241] Balance Scale Data Set . http://archive.ics.uci.edu/ml/datasets/balance+scale.

[242] Sharaf Malebary. DSRC Vehicle Communications Data Set . http://archive.ics.uci.edu/ml/datasets/DSRC+Vehicle+Communications.

[243] Nexar. the nexar dataset. https://www.getnexar.com/challenge-1/.

[244] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.

[245] Xinlei Pan, Yurong You, Ziyan Wang, and Cewu Lu. Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952*, 2017.

[246] A Geiger, P Lenz, C Stiller, and R Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[247] Facebook research. the babi dataset. https://research.fb.com/downloads/babi/.

[248] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[249] Stack Exchange Data Dump . https://archive.org/details/stackexchange.

[250] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

[251] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[252] California dmv failure reports. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/autonomousveh_ol316.

[253] Dr. Hans Hofmann. Statlog (german credit data) data set. http://archive.ics.uci.edu/ml/datasets/statlog+(german+credit+data).

[254] Ron Kohav. Adult data set. http://archive.ics.uci.edu/ml/datasets/adult.

[255] Sérgio Moro, Paulo Cortez, and Paulo Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22–31, 2014.

[256] Executions in the united states. https://deathpenaltyinfo.org/views-executions.

[257] Andrea Dal Pozzolo, Olivier Caelen, Reid A Johnson, and Gianluca Bontempi. Calibrating probability with undersampling for unbalanced classification. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 159–166. IEEE, 2015.

[258] Peter J Bickel, Eugene A Hammel, and J William O'Connell. Sex bias in graduate admissions: Data from Berkeley. *Science*, 187(4175):398–404, 1975.

[259] propublica. data for the propublica story 'machine bias'. https://github.com/propublica/compas-analysis/.

[260] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19, 2016.

[261] Zillow. Zillow Prize: ZillowâĂŹs Home Value Prediction (Zestimate). https://www.kaggle.com/c/zillow-prize-1/overview.

[262] US 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.

[263] Law School Admission Council. Lsac national longitudinal bar passage study (nlbps). http://academic.udayton.edu/race/03justice/legaled/Legaled04.htm.

[264] VirusTotal. Virustotal. https://www.virustotal.com/#/home/search.

[265] contagio malware dump. http://contagiodump.blogspot.com/2013/03/16800-clean-and-11960-malicious-files.html.

[266] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[267] Alen Shapiro. UCI chess (king-rook vs. king-pawn) data set. https://archive.ics.uci.edu/ml/datasets/Chess+%28King-Rook+vs.+King-Pawn%29, 1989.

[268] Anthony Finkelstein, Mark Harman, Afshin Mansouri, Jian Ren, and Yuanyuan Zhang. A search based approach to fairness analysis in requirements assignments to aid negotiation, mediation and decision making. *Requirements Engineering*, 14(4):231–245, 2009.

[269] Jie Zhang, Lingming Zhang, Dan Hao, Meng Wang, and Lu Zhang. Do pseudo test suites lead to inflated correlation in measuring test effectiveness? In *Proc. ICST*, pages 252–263, 2019.

[270] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[271] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

[272] Shin NAKAJIMA. Quality assurance of machine learning software. In *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pages 601–604. IEEE, 2018.