The Journal of Systems and Software 164 (2020) 110542

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On testing machine learning programs

Houssem Ben Braiek*, Foutse Khomh

SWAT Lab., Polytechnique Montréal, Canada

ARTICLE INFO

Article history:
Received 31 December 2018
Revised 15 December 2019
Accepted 6 February 2020
Available online 7 February 2020

Keywords:
Machine learning
Data cleaning
Feature engineering testing
Model testing
Implementation testing

ABSTRACT

Nowadays, we are witnessing a wide adoption of Machine learning (ML) models in many software systems. They are even being tested in safety-critical systems, thanks to recent breakthroughs in deep learning and reinforcement learning. Many people are now interacting with systems based on ML every day, *e.g.,* voice recognition systems used by virtual personal assistants like Amazon Alexa or Google Home. As the field of ML continues to grow, we are likely to witness transformative advances in a wide range of areas, from finance, energy, to health and transportation. Given this growing importance of ML-based systems in our daily life, it is becoming utterly important to ensure their reliability. Recently, software researchers have started adapting concepts from the software testing domain (*e.g.,* code coverage, mutation testing, or property-based testing) to help ML engineers detect and correct faults in ML programs. This paper reviews current existing testing practices for ML programs. First, we identify and explain challenges that should be addressed when testing ML programs. Next, we report existing solutions found in the literature for testing ML programs. Finally, we identify gaps in the literature related to the testing of ML programs and make recommendations of future research directions for the scientific community. We hope that this comprehensive review of software testing practices will help ML engineers identify the right approach to improve the reliability of their ML-based systems. We also hope that the research community will act on our proposed research directions to advance the state of the art of testing for ML programs.

1. Introduction

Machine learning (ML) is increasingly deployed in large-scale software systems thanks to recent breakthroughs in deep learning and reinforcement learning. We are now using software applications powered by ML in critical aspects of our daily lives; from finance, energy, to health and transportation. However, detecting and correcting faults in ML programs is still very challenging as evidenced by the recent incident with an Uber autonomous car that resulted in the death of a pedestrian (Gibbs, 2018). The main reason behind the difficulty to test ML programs is the shift in the development paradigm induced by ML and AI. Traditionally, software systems are constructed deductively, by writing down the rules that govern the behavior of the system as program code. However, with ML, these rules are inferred from training data (*i.e.,* they are generated inductively). This paradigm shift in application development makes it difficult to reason about the behavior of software systems with ML components, resulting in systems that are intrinsically challenging to test and verify, given that they do not have (complete) specifications or even source code corresponding to some of their critical behaviors. In fact some ML programs rely on proprietary third-party libraries like Intel Math Kernel Library for many critical operations. A defect in a ML program may come from its training data, program code, execution environment, or third-party frameworks. Compared with traditional software, the dimension and potential testing space of a ML programs is much larger. Current existing software development techniques must be revisited and adapted to this new reality.

In this paper, we survey existing testing practices that have been proposed for ML programs, explaining the context in which they can be applied and their expected outcome. We also identify gaps in the literature related to the testing of ML programs and suggest future research directions for the scientific community. This paper makes the following contributions:

- We present and explain challenges related to the testing of ML programs that use differentiable models.
- We provide a comprehensive review of current software testing practices for ML programs.
- We identify gaps in the literature related to the testing of ML programs and provide future research directions for the scientific community.

* Corresponding author.
E-mail addresses: houssem.ben-braiek@polymtl.ca (H.B. Braiek), foutse.khomh@polymtl.ca (F. Khomh).

https://doi.org/10.1016/j.jss.2020.110542
0164-1212/

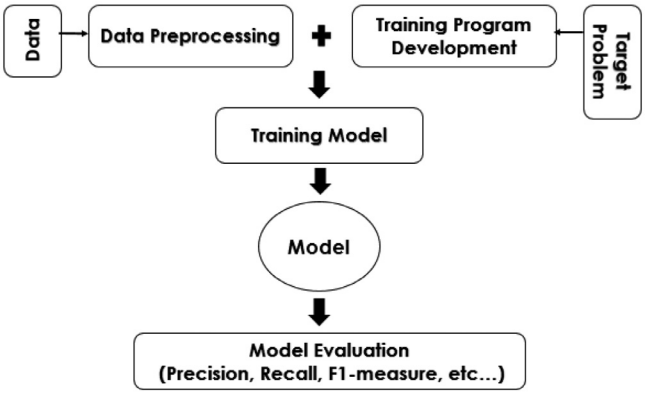

Fig. 1. Basic procedure of ML model construction.

To the best of our knowledge, this is the most comprehensive study of testing practices for ML programs. **The rest of the paper is organized as follows.** Section 2 provides background knowledge on ML programs. Section 3 presents research trends in ML application testing. Section 4 summarizes the approaches proposed for testing and debugging the data preprocessing tasks while Section 5 summarizes the approaches proposed for testing the ML model and debugging the training program implementation. Next, Section 6 outlines some avenues for future works. Section 7 discusses related surveys from the literature. Finally, Section 8 concludes the paper.

2. Background on machine learning programs

In this section, we provide background information about Machine Learning (ML) programs and explain challenges that should be addressed when testing ML programs.

The first step when constructing a ML component is to collect data from which concepts and hidden patterns can be learned using some algorithms. Most machine learning algorithms require huge volume of data to be able to converge and make meaningful inferences, which makes data collection a challenging step. Once data is collected, often it has to be pre-processed before it can be used for learning. Failing to complete this pre-processing properly is likely to result in noisy data which can significantly affect the quality of trained models. After this pre-processing step, important features are identified from the data. These features also often need to be processed before they can be used in a learning algorithm. Inadequate feature engineering (*i.e.,* failure to process the features adequately) is also likely to result in poor models. Once the data is cleaned and features are extracted properly, a learning algorithm is used to infer relations capturing hidden patterns in the data. During this learning process, the parameters of the algorithm are tuned to fit the input data through an iterative process, during which the performance of the model is assessed continuously. A poor choice of parameter or an ineffective model testing mechanism will also result in a poor model. After training and testing steps, the model is deployed in a production environment which can be different from the training and testing environments. In production, the model often has to interact with the other components of the application. Fig. 1 presents the basic procedure of building an ML model.

There are two main source of faults in ML programs : the data and the model. For each of these two dimensions (*i.e.,* data and model), there can be errors both at the conceptual and implementation levels, which makes fault location very challenging. Approaches that rely on tweaking variables and watching signals from execution are generally ineffective because of the exponential increase of the number of potential faults locations. In fact, if there are n potential faults related to data, and m potential faults related to the model, we have $n \times m$ total possible faults locations when data is fed in the model. This number grows further when the model is implemented as code, since more types of faults can also be introduced in the program at that stage. Systematic and efficient testing practices are necessary to help ML engineers detect and correct faults in ML programs. In the following, we present potential errors that can occur in data and models, both at design and implementation levels.

2.1. Data engineering: Challenges and issues

Data is an essential artifact when building ML models. It often comes from a variety of sources *e.g.,* mainframe databases, sensors, IoT devices, and software systems, and is presented in different formats (*e.g.,* various media types). It can be structured (such as database records) or unstructured (such as raw text) and is delivered to ML models either in batch (*e.g.,* discrete chunks from mainframe databases and file systems) and/or real-time (*e.g.,* continuous flow from IoT devices or Stream REST APIs). ML engineers generally have to leverage complementary automated tools that support batch and/or real-time data ingestion strategies, to collect data needed for training ML models.

Conceptual issues Once data is gathered, data cleaning tasks are required to ensure that the data is consistent, free from redundancy and given a reliable starting point for statistical learning. Common cleaning tasks include: (1) removing invalid or undefined values (*i.e.,* Not-a-Number, Not-Available), duplicate rows, and outliers that seems to be too different from the mean value); and (2) unifying the variables' representations to avoid multiple data formats and mixed numerical scales. This can be done by data transformations such as normalization, min-max scaling, and data format conversion. This pre-processing step ensures a high quality of raw data, which is very important because decisions made on the basis of noisy data could be wrong. In fact, recent sophisticated ML models endowed by a high learning capacity are highly sensitive to noisy data (Krishnan et al., 2016). This brittleness makes model training unreliable in the presence of noisy data, which often results in poor prediction performances (Qi et al., 2018). A good illustration of this issue is provided by McDaniel et al. (2016), who showed that the nonlinearity and high learning capacity of DNN models allow them to construct boundary decisions that conform more tightly to their training data points. In the absence of carefully crafted regularization techniques, these DNNs tend to overfit their training data instead of learning robust high-level representations.

Conventional ML algorithms (e.g., logistic regression, support vector machine, decision tree... etc) require feature engineering model methods and considerable domain expertise to perform the extraction of a vector of semantically useful features from raw data that allow the ML algorithm to detect relevant patterns and learn an explainable mapping function.

In fact, the performance of these machine learning algorithms depends heavily on the representation of the data that they are given. Pertinent features that describe the structures inherent in the data entities need to be selected from a large set of initial variables in order to eliminate redundant or irrelevant features. This selection is based on statistical techniques such as correlation measures and variance analysis. Afterwards, these features are encoded to particular data structures to allow feeding them to a chosen ML model that can handle the features and recognize their hidden related patterns in an effective way. The identified patterns represent the core logic of the model. Changes in data (*i.e.,* the input signals) are likely to have a direct impact on these patterns and hence on the behavior of the model and its corresponding

predictions. Because of this strong dependence on data, ML models are considered to be data-sensitive or data-dependent algorithms. It is also possible to define features manually. However, these human-crafted features often require a huge amount of time and effort, and like any informal task, they can be subject to human errors and misunderstandings. A poor selection of features can impact a ML system negatively. Sculley et al. (2015) report that unnecessary dependencies to features that contribute with little or no value to the model quality can generate vulnerabilities and noises in a ML system. Examples of such features are : *Epsilon Feature*, which are features that have no significant contribution to the performance of the model, *Legacy Feature*, which are features that lost their added information value on model accuracy improvement when other more rich features are included in the model, or *Bundled Features*, which are groups of features that are integrated to a ML system simultaneously without a proper testing of the contribution of each individual feature.

In the case of deep learning, feature inference is done automatically through a cascade of multiple representation levels, where higher-level abstractions are defined in terms of lower-level ones. This powerful computational model includes multiple processing layers that uncover unknown patterns in a raw data distribution in order to identify relevant hierarchy of features and learn complex mapping functions for a target prediction. However, recent works Jo and Bengio (2017) show that DNN, without carefully tuned hyperparameters and a healthy training process, tend to learn superficial regularity instead of learning high-level abstract features that are less likely to overfit the data. **Implementation issues** To process data as described above, ML engineers implement data pipelines containing components for data transformations, validation, enrichment, summarization, and/or any other necessary treatment. Each pipeline component is separated from the others, and takes in a defined input, and returns a defined output that will be served as input data to the next component in the pipeline. Data pipelines are very useful in the training phase as they help process huge volume of data. They also transform raw data into sets of features ready-to-be consumed by the models. Like any other software component, data pipelines are not immunized to faults. There can be errors in the code written to implement a data pipeline. Sculley et al., 2014 identified two common problems affecting data pipelines:

- *Pipelines jungles* that are overly convoluted and unstructured data preparation pipelines. This appears when data is transformed from multiple sources at various points through scraping, table joins and other methods without a clear, holistic view of what is going on. Such implementation is prone to faults since developers lacking a good understanding of the code are likely to make mistakes. Also, debugging errors in such code is challenging.
- *Dead experimental code paths* that happen when code is written for rapid prototyping to gain quick turnaround times by performing additional experiments simply by tweaks and experimental code paths within the main production code. The remnants of alternative methods that have not been pruned from the code base could be executed in certain real world situations and create unexpected results.

For systems that rely on mini-batch stochastic optimizers to estimate the model's parameters, like deep learning models, another data-related component is required in addition to data pipelines, *i.e.,* the Data Loader. This component is responsible of the generation of the batches that are used to assess the quality of samples, during the training phase. It is also prone to errors.

2.2. Model engineering: Challenges and issues

Once ML engineers have collected and processed the data, they proceed to finding the appropriate statistical learning model that could fit the available data in order to build its own logic and solve the given problem. A wide range of statistical models can be acquired and/or extended to suit different classification and regression purposes. There are simple models that make initial assumptions about hidden relationships in the data. For example, the linear regression assumes that the output can be specified as a linear combination of features and the SVM classifier assumes that there is an hyperplane with maximum margin that can optimally separate the two classes of the output. Besides, there are more complex models such as Neural Network models, which usually do not make assumptions about the relationship between incoming pairs of data. In fact, a Neural Network is structured in terms of interconnected layers of computation units, much like the neurons and their connectivity in the brain. Each neuron includes an activation function (*i.e.,* a non-linear transformation) to a weighted sum of input values with bias. The predicted output is calculated by computing the outputs of each neuron through the network layers in a feed-forward manner. At the end, a Neural Network's mapping function can be seen as a composite function encoding a sequence of linear transformations and their following non-linear ones. The strength of this complex model lies in the universal approximation theorem. A feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous function, under mild assumptions on the activation function. However, this does not indicate how much neurons are required and the number can evaluate exponentially with respect to the complexity of formulating the relationships between inputs and outputs. Deep neural networks, which are the backbone behind deep learning, use a cascade of multiple hidden layers to reduce the number of neurons required in each layer.

Regardless of the model, ML programs discover hidden patterns in training data and build a mathematical model that makes predictions or identifications on future unseen data, using these patterns. The learning aspect resides in the model fitting, which is an iterative process during which little adjustments are made repeatedly, with the aim of refining the model until it predicts mostly the right outputs. Generally, supervised machine learning algorithms are based on a differentiable model that trains itself on input data through optimization routines using gradient learning, to create a better model or probably the best fitted one (*i.e.,* this could happen with a convex objective function or advanced update steps). The principal components of a differentiable model are:

- **Parameters**, on which the model depends to make its internal calculation and to provide its prediction outputs. Therefore, these signals or factors are inner variables of the model that the ML program gradually adjusts on its own through successive training iterations to build its logic and form its decision about future data. For example, weights and biases used by simple linear regression models or by Neural Network's neurons are parameters.
- **Loss Function**, at its core, represents a mathematical function given a real value that provide an estimation on how the model is performing in terms of learning goals. It assesses the sum or the average of the distance measure between predicted outputs and actual outcomes. As indicated by its name, it reflects the cost or the penalty of making bad predictions. If the model's predictions are perfect, the loss is zero; otherwise, the loss is greater.
- **Regularization**, assembles techniques, that penalize the model's complexity to prevent overfitting. An example of regularization technique is the use of *L2* norm in linear regression

models, which keeps smaller overall weight values, relying on a prior belief that weights should be small and normally distributed around zero. Another example of regularization technique is the *dropout* in neural networks, which allows removing a random selection of a percentage of neurons from training during an iteration. This is to prevent models with high learning capacity from memorizing the peculiarities of the training data, which would result in complex models that overfit the training data. To guarantee a model's generalization ability, two parallel objectives should be reached: (1) build the best-fitted model *i.e.,* lowest loss and (2) keep the model as simple as possible *i.e.,* strong regularization.

- **Optimizer**, adjusts iteratively the internal parameters of the model in a way that reduces the objective function, *i.e.,* includes generally the loss function plus a regularisation term. The most used optimizers are based on gradient descent algorithms, which gradually minimize the objective function by computing the gradients of loss with respect to the model's parameters and updates their values in direction opposite to the gradients until finding a local or the global minimum. The objective function has to be globally continuous and differentiable. It is desirable that this function be also strictly convex and has exactly one local minimum point, which is also the global minimum point. A great deal of research in ML has focused on formulating various problems as convex optimization problems and solving them efficiently. Deep neural networks are never convex functions but they are very successful because many variations of gradient descent have a high probability of finding reasonably good solutions anyway, even though these solutions are not guaranteed to be global minimums.
- **Hyperparameters**, represent the model's parameters that are constant during the training phase and which can be fixed before running the fitting process. By learning a ML model, the ML engineers identify a specific point in the model space with some desirable behavior. In fact, choosing in advance the hyperparameters of the model, such as the number of layers and neurons in each layer for neural network, the learning rate for gradient descent optimizer or the regularization rate, allows to identify a subset of the model space to search. Finally, the optimizer must be used to find the best-fit model from the selected subset via parameter adjustments. Similarly, hyperparameters tuning is highly recommended to find the composition of hyper-parameters that helps the optimization process finding the best-fitted model. The most used search methods are (1) *Grid search*, which explores all possible combinations from a discrete set of values for each hyperparameter; (2) *Random search* which samples random values from a statistical distribution for each hyperparameter, and (3) *Bayesian optimization* which chooses iteratively the optimal values according to the posterior expectation of the hyperparameter. This expectation is computed based on previously evaluated values, until converging to an optimum.

To train a differentiable model one needs three different datasets : *training dataset, validation dataset*, and *testing dataset*. After readying these data sets, ML engineers set initial hyperparameters values and select loss functions, regularization terms, and gradient-based optimizers following best practices or guidelines from other works that addressed a similar problem. Training a ML model using an optimizer consists in gradually minimizing the loss measure plus the regularization term with respect to the training dataset. Once the model's parameters are estimated, hyperparameters are tuned by evaluating the model performance on the *validation dataset*, and selecting the next hyperparameter values according to a search-based approach that aims to optimize the performance of the model. This process is repeated using the newly selected hyperparameters until a best-fitted model is obtained. This best-fitted model is therefore tested using the testing dataset (which should be different from training and validation datasets).
Conceptual issues One key assumption behind the training process of supervised ML models is that the *training dataset*, the *validation dataset*, and the *testing dataset*, which are sampled from manually labeled data, are representative samples of the underlying problem. Following the concept of Empirical Risk Minimization (ERM), the optimizer finds the fitted model that minimizes the empirical risk; which is the loss computed over the training data assuming that it is a representative sample of the target distribution. The empirical risk can correctly approximates the *true risk* only if the training data distribution is a good approximation of the true data distribution (which is often out of reach in real-world scenarios). The size of the training dataset has an impact on the approximation goodness of the true risk, *i.e.,* the larger a training data, the better this approximation will be. However, manual labeling of data is very expensive, time-consuming and error-prone. Training data sets that deviate from the reality induce erroneous models. The configuration of model includes the hyperparameters that control its capacity (such as number of hidden layers or the depth of decision tree) and also control the behavior of the training algorithm (such as the number of epochs or learning rate). A poor choice of hyperparameters, often results in models with poor performance. For example, inappropriate capacity may result in the model capturing irrelevant information, i.e., noise (overfitting) or missing important data features (underfitting). **Implementation issues** ML algorithms are often proposed in pseudo-code formats that include jointly scientific formula and algorithmic rules and concepts. When it comes to implementing ML algorithms, ML engineers sometimes have difficulties understanding these formulas, rules, or concepts. Moreover, because there is no "test oracle" to verify the correctness of the estimated parameters (*i.e.,* the computation results) of a ML model, it is difficult to detect faults in the learning code. Also, ML algorithms often require sophisticated numerical computations that can be difficult to implement on recent hardware architectures that offer high-throughput computing power. Weyuker (1982) identified three distinct sources of errors in scientific software programs such as ML programs.

1. **The mathematical model used to describe the problem**
   A ML program is a software implementation of a statistical learning algorithms that requires substantial expertise in mathematics (*e.g.,* linear algebra, statistics, multivariate analysis, measure theory, differential geometry, topology) to understand its internal functions and to apprehend what each component is supposed to do and how we can verify that they do it correctly. Non-convex objectives can cause unfavorable optimization landscape and inadequate search strategies. Model mis-specifications or a poor choice of mathematical functions can lead to undesired behaviors. For example, many ML algorithms require mathematical optimizations that involve extensive algebraic derivations to put mathematical expressions in closed-form. This is often done through informal algebra by hand, to derive the objective or loss function and obtain the gradient. Generally, we aim to adjust model parameters following the direction that minimizes the loss function, which is calculated based on a comparison between the algorithmic outputs and the actual answers. Like any other informal tasks, this task is subject to human errors. The detection of these errors can be very challenging, in particular when randomness is involved. Errors in stochastic computations can persist indefinitely without detection, since some of them may be masked by the distributions of random variables and may require writing customized statistical tests for their detection.

To avoid overfitting, ML engineers often add a regularization loss (*e.g.,* norm L_2 penalty on weights). This regularization term which has a simple gradient expression can overwhelm the overall loss; resulting in a gradient that is primarily coming from it. Which can make the detection of errors in the real gradient very challenging. Also, non-deterministic regularization techniques, such as dropout in neural network can cause high-variance in gradient values and further complicate the detection of errors, especially when techniques like numerical estimation are used.

2. **The program written to implement the computation**
The program written to implement a mathematical operation can differ significantly from the intended mathematical semantic when complex optimization mechanisms are used. Nowadays, most ML programs leverage rich data structures (*e.g.,* data frames) and high performance computing power to process massive data with huge dimensionality. The optimization mechanisms of these ML programs is often solved or approximated by linear methods that consists of regular linear algebra operations involving, for example, multiplication and addition operations on vectors and matrices. This choice is guided by the fact that high performance computers can leverage parallelization mechanisms to accelerate the execution of programs written using linear algebra. However, to leverage this parallelism on Graphics Processing Unit (GPU) platforms for example, one has to move to higher levels of abstraction. The most common abstractions used in this case are tensors, which are multidimensional arrays with some related operations. Most ML algorithms nowadays are formulated in terms of matrix-vector, matrix-matrix operations, and tensor-based operations (to extract a maximum of performance from the hardware). Also, ML models are more and more sophisticated, with multiple layers containing huge number of parameters each. For such models, the gradient which represents partial derivatives that specify how the loss function is altered through individual changes in each parameter is computed by grouping the partial derivatives together in multidimensional data structures such as tensors, to allow for more straight forward optimized and parallelized calculations. This large gap between the mechanics of the high performance implementation and the mathematical model of one ML algorithm makes the translation from scientific pseudocode to highly-optimized program difficult and error-prone. It is important to test that the code representations of these algorithms reflect the algorithms accurately.
3. **Features of the environment, such as round-off error**
The computation of continuous functions such as gradient on discrete computational environments like a digital computer incurs some approximation errors when one needs to represent infinitely many real numbers with a finite number of bit patterns. These numerical errors of real numbers' discrete representations can be either overflow or underflow. An overflow occurs when numbers with large magnitude are approximated as $+\infty$ or $-\infty$, which become not-a-number values if they are used for many arithmetic operations. An underflow occurs when numbers near zero are rounded to zero. This can cause the numerical instability of functions such as division (*i.e.,* division by a zero returns not-a-number value) or logarithm (*i.e.,* logarithm of zero returns $-\infty$ that could be transformed into not-a-number by further arithmetic).
Hence, it is not sufficient to validate a scientific computing algorithm theoretically since rounding errors can lead to the failure of its implementation. Rounding errors on different execution platforms can cause instabilities in the execution of ML models if their robustness to such errors is not handled properly. Testing for these rounding errors can help select adequate mathematical formulations that minimize their effect.

When implementing ML programs, developers often rely on third-party libraries for many critical operations. These libraries provide optimized implementations of highly intensive computation functions, allowing ML developers to leverage distributed infrastructures such as high-performance computing (HPC) clusters, parallelized kernels executing on high-powered graphics processing units (GPUs), or to merge these technologies to breed new clusters of multiple GPUs. However, a misuse of these libraries can result in faults that are hard to detect. For example, a copy-paste of a routine that creates a neural network layer, without changing the corresponding parameters (*i.e.,* weights and biases variables) would result in a network where the same parameters are shared between two different layers, which is problematic. Moreover, this error can remain in the code unnoticed for a long time. This is why it is utterly important to test that configuration choices do not cause faults or instabilities in ML programs. In the following, we explain the three main categories of libraries that exist today and discuss their potential misuse.

- **High-level ML libraries:** A high-level ML library emphasizes the ease of use through features such as state-of-the-art supervised learning algorithms, unsupervised learning algorithms, and evaluation methods (*e.g.,* confusion matrix and cross-validation). It serves as an interface and provides a high level of abstraction, where ML algorithms can be easily configured without hard-coding. Using such libraries, ML developers can focus on converting a business objective into a ML-solvable problem. However, ML developers still need to test the quality of data and ensure that it conforms to the requirements of these built-in functions, such as input data formats and types. Moreover, a poor configuration of these provided algorithms could result in unstable or misconceived models. For example, choosing a sigmoid as an activation function in a deep neural network can cause the saturation of neurons and consequently, slow down the learning process. Therefore, after finishing the configuration, developers need to set up monitoring routines to watch internal variables and metrics during the execution of the provided algorithms, in order to check for possible numerical instabilities, or suspicious outputs.
- **Medium-level ML libraries:** Medium-level libraries provide machine learning or deep learning routines as ready-for-use features, such as numerical optimization techniques, mathematical function and automatic differentiation capabilities, allowing ML developers to not only configure the pre-defined ML algorithms, but also to use the provided routines to define the flow of execution of the algorithms. This flexibility allows for easy extensions of the ML models using more optimized implementations. However, this ability to design the algorithm and its computation flow through programming variables and native loops increases the risk of faults and poor coding practices, as is the case for any traditional program.
- **Low-level ML libraries:** Contrary to high-level and medium-level libraries, this family of libraries do not provide any predefined ML feature, instead, they provide low-level computations primitives that are optimized for different platforms. They offer powerful *N*-dimensional arrays, which are commonly used in numerical operations and linear algebra routines for a high level of efficiency. They also help with data processing operations such as slicing and indexing. ML developers can use these libraries to build a new ML algorithm from scratch or highly-optimized implementations of particular algorithms for specific contexts or hardwares. However, this total control on the implementation is not without cost. Effective quality assurance operations such as testing and code reviews are required to ensure bug free implementations. ML developers need strong

backgrounds in mathematics and programming to be able to work efficiently with these low level libraries.

The amount of code that is written when implementing a ML program depends on the type of ML library used. The more a developer uses high-level features from libraries, the more he has to write glue code to integrate incompatible components, which are putted together into a single implementation. Sculley et al., 2014 observed that the amount of this glue can account for up to 95% of the code of certain ML programs. This code should be tested thoroughly to avoid faults.

3. Review of the literature on testing ML applications

To identify research works that proposed testing techniques for ML programs, we constructed a repository of 1305 publications by conducting web searches for research articles on Engineering village, Google Scholar, and Microsoft Academic Search, using queries consisting of three types of keywords, *i.e.,* "a ML-related keyword" + "a testing-related keyword" + " a testing dimension-related keyword". Examples of ML-related keywords includes "Machine Learning", "ML", "deep learning", "neural network" or popular applications such as "image classifier" or "autonomous cars". Testing-related keywords includes "test", "error", "bug", "verification", while testing dimension-related keywords indicate the dimension being tested, *e.g.,* "data", "model", "application", "software", "program", "system" and "implementation". Next, we performed a mapping study (also referred to as scoping review) in order to identify and categorise these primary studies based on their scope. This scoping review helps identify the main issues addressed and studied in the available literature. Next, we selected the most relevant studies based on their titles and abstracts. Any irrelevant study was removed. If there were any doubt about any study at this level, the study was kept. All the remaining studies at this step were carefully read. We applied a backward snowball search using these remaining studies to identify new papers and studies. As a result of this selection process, we retained 37 publications that reported approaches for testing ML programs. These papers are published between 2007 and 2019. Fig. 3 shows the evolution of the publication count per year. We can see a sharp increase in research works that focused on testing ML-based systems in the last years, which can be explained by the growing number of real-world applications of deep learning. We organised these publications in three groups, each of them representing one grey-box step in the Fig. 2.

This classification considers the component under test and the objective of test. The first group contains papers that aim to detect conceptual and implementation errors in data. The second group contains papers that aim to detect conceptual and implementation errors in ML models. The third group contains papers that consider the training program as the component under test and aim to find implementation errors; including programming bugs, computation mistakes and numerical instabilities. In total, we have 5 papers from the first group, 19 papers from the second group, and 13 papers from the third group.

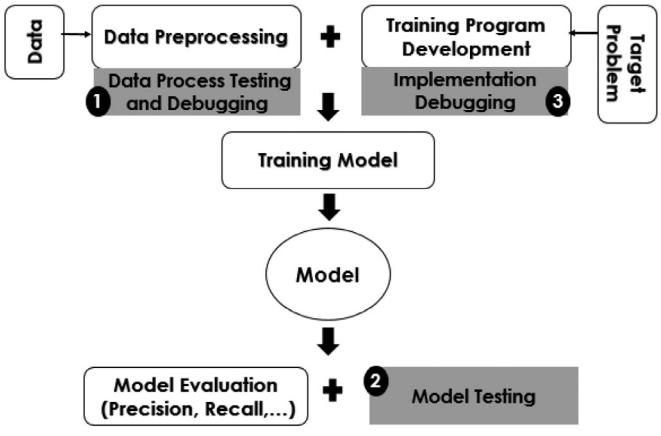

Fig. 2. Testing and debugging phases in ML development workflow.

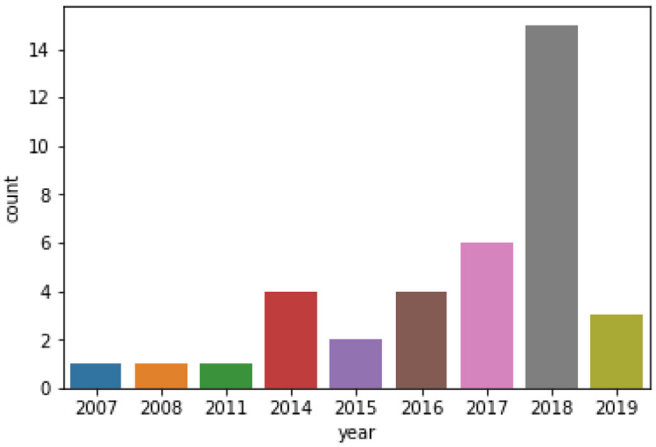

Fig. 3. The number of publications per year.

In each of these categories, we divide the techniques in subgroups based on the concepts used in the techniques. Next, we discuss the fundamental concepts behind each proposed technique, explaining the types of errors that can be identified using them while also outlining their limitations.

4. Approaches that aim to detect conceptual and implementation errors in data

The approaches proposed in the literature to test the quality of data address both conceptual and implementation issues, therefore, we discuss these two aspects together in this section. The most common technique used to test the quality of data is the analysis-driven data cleaning that consists of applying analytical queries (*e.g.,* aggregates, advanced statistical analytic, etc.) in order to detect errors and perform adequate transformations.

In this approach, aggregations such as sum or count and central tendencies such as mean, median or mode are used to verify if each feature's distribution matches expectation. For example, one can check that features take on their usual set or range of values, and the frequencies of these values in the data.

Qi et al. (2018) investigated the effects of missing, inconsistent and conflicting data on the performance of ML algorithms using 13 well-known datasets. This study was conducted by injecting errors into the underlying data, and then, assessing the quality of models trained on that data. They observed that dirty data consistently had a negative impact on the performance (*i.e.,* Precision/Recall/F-measure) of the studied models. However, the magnitude of the impact depended on both the error type and the error rate. They also observed that increases in the size of data reduce the negative effect of induced errors. These results encourage the use of big data to reduce the impact of data errors and inconsistencies on the trained model.

This feature engineering of training data should not be confused with the feature engineering of models, which consists of creating relevant features for ML models based on domain knowledge and expert intuition. The feature engineering of the model is an essential step in the construction of conventional ML models. However,

in the case of DL models, the features are inferred automatically. In fact, DL models build complex features automatically as a part of their statistical learning process from data. For example, conventional computer-vision models require image features, including edges, corners and blobs that can be detected using low-level image processing operations, while Convolutional Neural Networks process raw images directly. We do not discuss the quality assurance of feature engineering of models because it is often tightly connected to the type of models and/or the application context; *i.e.,* for each domain, experts often use specific metrics to evaluate the quality of their extracted features.

For conventional ML models, once features are identified, advanced statistical analyses such as hypothesis testing and correlation analysis are applied to verify correlations between pair of features and to assess the contribution of each feature in the prediction or explanation of the target variable. The benefit of each feature can also be estimated by computing the proportion of its explained variance with respect to the target output or by assessing the resulting accuracy of the model when removing it in prior to the fitting process. Besides, when assessing the contribution of each feature to the model, it is recommended to take into account the added inference latency and RAM usage, more upstream data dependencies, and additional expected instability incurred by relying on that feature. It is important to consider whether this cost is worth paying when traded off against the provided improvement in model quality.

Recent research work by Krishnan et al. (2015) remarked that these aggregated queries can sometimes diminish the benefits of data cleaning. They observed that cleaning small samples of data often suffices to estimate results with high accuracy. They also observed that the power of statistical anomaly detection techniques rapidly deteriorates in the high-dimensional feature-spaces.

This comes from the fact that the aforementioned analysis-driven data cleaning operations require data queries in order to calculate the aggregate values and correlation measures. Indeed, performing many data queries and cleaning operation on the entire dataset could be impractical with huge amount of training datasets that likely contain dirty records. Moreover, ML developers often face difficulties in establishing the data cleaning process. To address these issues, Krishnan et al. proposed ActiveClean (Krishnan et al., 2016), an interactive data-cleaning framework for ML models that allows ML developers to improve the performance of their model progressively as they clean the data. The framework has an embedding process that firstly samples a subset of likely dirty records from training data using a set of optimizations, which includes importance weighting and dirty data detection. Secondly, the ML developer is interactively invited to transform or remove each data instance from the selected batches of probably dirty data. Finally, the framework updates the model's parameters and continues the training using partially cleaned data. This process is repeated until no potential dirty instances could be detected. With ActiveClean, developers are still responsible for defining data cleaning operations. The framework only decides where to apply these operations. Recently, Krishnan et al. (2017) proposed a fully automated framework, BoostClean, to establish a pipeline of data transformations that allow cleaning efficiently the data in order to fit well the model. BoostClean automatically finds the best combination of dirty data detection and repair operations by leveraging the available clean test labels in order to improve model accuracy. It selects this ensemble from extensible libraries : (1) pre-populated general detection functions, allow identifying numerical outliers, invalid and missing values, checking whether a variable values match the column type signature, and detecting effectively text errors in string-valued and categorical attributes using word embedding; (2) a pre-populated set of simple repair functions that can be applied to records identified by a detector's predicate, such as impute a cell value with one central tendency (*i.e.,* mean, median and mode value), or discard a dirty record from the dataset. Thus, the boosting technique, which combines a set of weak learners and estimates their corresponding weights to spawn a single prediction model, is applied to solve the problem of detecting the optimal sequence of repairs that could best improve the ML model by formulating it as an ensemble learning problem. It consists of generating a new model trained on input data with new additional cleaned features and selecting the best collection of models that collectively estimate the prediction label. Krishnan et al. evaluated their proposed framework on 8 ML datasets from Kaggle and the UCI repository which contain real data errors. They showed that BoostClean can increase the absolute prediction accuracy by 8–9% over the best non-ensemble alternatives, including statistical anomaly detection and constraint-based techniques.

By automating the selection of cleaning operations, BoostClean significantly simplifies the data cleaning process, however, this framework is resource consuming as it requires the use of multiple models and boosting techniques. Moreover, the evaluation of the embedded cleaning process results on new datasets is challenging because the creation of the data cleaning pipeline is driven by pure statistical analysis.

Hynes et al., 2017, inspired by code linters, which are well-known software quality tools, introduced data linter to help ML developers track and fix issues in relation to data cleaning, data transformation and feature extraction. The data linter helps reduce the human burden by automatically generating issues explanations and building more sophisticated human-interactive loop processing. First, it inspects errors in training datasets such as scale differences in numerical features, missing or illegal values (*e.g.,* NaN), malformed values of special string types (*e.g.,* dates), and other problematic issues or inefficiencies discussed in Section 2.1. The inspection relies on data's summary statistics, individual items inspection, and column names given to the features. Second, given the detected errors and non-optimal data representations, it produces a warning, a recommendation for how to transform the feature into a correct or optimal feature, and a concrete instance of the lint taken directly from the data. Data linter guides its users in their cleaning data and features engineering process through providing actionable instructions of how individual features can be transformed to increase the likelihood that the model can learn from them. The main strength of this tool resides in the semi-automated data engineering process and the fact that it can be applied to all statistical learning models and several different data types. The proposed data linter has the ability to infer semantic meaning/intent of a feature based on its metadata as a complement to statistical analysis, with the aim of providing specific and comprehensible feature engineering recommendations to ML developers. For example, using a data linter, a developer can automatically discern whether a string or a numeric data represents a zip code or not. An information that is difficult to obtain by simply inspecting raw values. As mentioned in Section 2.1, a Data loader is often required for systems that rely on mini-batch stochastic optimizers to estimate the model's parameters. To test the reliability of this component, the following best practices are often used: (1) Shuffling of the dataset before starting the batches generation. This action is recommended to prevent the occurrence of one single label batch (*i.e.,* sample of data labeled by the same class) which would negatively affect the efficiency of mini-batch stochastic optimizers in finding the optimal solution. In fact, a straightforward extraction of batches in sequence from data ordered by label or following a particular semantic order, can cause the occurrence of one single label batch. (2) Checking the predictor/predict inputs matching. A random set of few inputs should be checked to verify if they are

correctly connected to their labels following the shuffling of data. (3) Reduce class imbalance. This step is important to keep the class proportions relatively conformed to the totality of training data.

5. Approaches that aim to detect conceptual and implementation errors in ML models

As discussed in Section 2.2, errors in ML models can be due to conceptual mistakes when creating the model or implementation errors when writing the code corresponding to the model. In the following, we discuss testing approaches that focus on these two aspects, separately.

5.1. Approaches that aim to detect conceptual errors in ML models

Approaches in this category assume that the models are implemented into programs without errors and focus on providing mechanisms to detect potential errors in the calibration of the models. These approaches can be divided in two groups: black-box and white-box approaches (Nidhra and Dondeti, 2012). Black-box approaches are testing approaches that do not need access to the internal implementation details of the model under test. These approaches focus on ensuring that the model under test predicts the target value with a high accuracy, without caring about its internal learned parameters. White-box testing approaches, on the other hand, take into account the internal implementation logic of the model. The goal of these approaches is to cover a maximum of specific spots (*e.g.,* neurons) in the models. In the following, we elaborate more on approaches from these two groups.

5.1.1. Black-box testing approaches for ML models

The common denominator to black-box testing approaches is the generation of an adversarial data set that is used to test the ML models. These approaches leverage statistical analysis techniques to devise a multidimensional random process that can generate data with the same statistical characteristics as the input data of the model. **Generative Models** These ML models are constructed to fit a probability distribution that best describes the input data. Indeed, they sample the probability distribution of input data and generate as many data points as needed for testing the under test ML model. Using the generative models, the input data set is slightly perturbed to generate novel data that retains many of the original data properties. The advantage of this approach is that the synthetic data that is used to test the model is independent from the ML model, but statistically close to its input data. One prominent type of generative model are Generative adversarial networks (GANs) Goodfellow et al. (2014). Over the last decade, GANs have been proven to outperform other types of generative models such as variational auto-encoders (VAEs) Doersch (2016) or restricted Boltzmann machines (RBMs) Fischer and Igel (2012). GANs are generative learning approaches that assemble two separate DNNs, a generator and discriminator, with opposing or adversarial objectives. The discriminator is trained to distinguish between original and synthetic samples, while the generator is trained to fool the discriminator through realistic synthetic data. Although GANs have been successfully trained on high dimensional continuous data to generate diverse and realistic examples, they cannot be applied when original datasets are discrete such as words, characters, or bytes. Recently, the boundary-seeking GAN (BGAN) Hjelm et al. (2017) has been proposed as a unified generative learning method that enables the generation of discrete data and improves the stability of the training on continuous data. A key characteristic of BGANs is that they reinterpret the generator objective as a distance instead of divergence. In fact, most common difference measures used by GANs come from the family of f-divergences (such as the KL-divergence), while BGANs define a new objective for the generator that represents a simple distance between log-probabilities of discriminator's outputs. This way, the generator learns to minimize this distance in order to make the discriminator output both real and fake labels with the same probability. GANs and BGANs have been successfully used to change data contexts and infer realistic situations from samples of training data, in order to test the robustness of the representations learned by models (Zhang et al., 2018a). **Adversarial ML: Alteration of existing inputs** When ML models are involved in security-sensitive applications, their robustness against non obvious and potentially dangerous manipulation of inputs should be tested. ML models can be vulnerable to malicious adaptive adversaries that manipulate their input data; causing them to diverge from the training data. In fact, the training data cannot cover the entire input or features space, especially, when dealing with high-dimensional data. This makes it difficult to approximate the real decision boundaries, since ML algorithms learn by minimizing the empirical risk on training data. This phenomenon becomes more nuanced for sophisticated models with high capacity, such as DNNs, which are able to draw complex decision boundaries with original shapes in order to conform more tightly to the data points and reduce the error as much as possible. This complexity creates vulnerabilities that can be exploited by adversaries. Adversarial machine learning is an emerging area where various techniques are being used to find adversarial regions where models exhibit erroneous behaviors. Several mechanisms exist for the creation of adversarial examples. Goodfellow et al. (0000) proposed a gradient-based perturbation that makes small modifications to drive the mutated input into ambiguous and vulnerable regions of the input space. It consists of computing, first, the gradient of the loss with respect to the input, to determine a suitable direction of changes. Then, it perturbs inputs towards this direction using a prefixed step size to control the magnitude of the perturbations. Engstrom et al. (2017) introduced affine transformations (such as translations and rotations) that can fool DNN-based vision models. Contrary to gradient-based perturbations, those affine transformations do not require any complicated optimization technique, but they are easy to find using a few black-box queries on the target model. A recent research work (Gilmer et al., 2018) shows that adversarial examples can be found through simple guess-and-check of naturally-occurring situations related to the application domain. This requires human effort and time, to produce realistic inputs. Recent results (Gu and Rigazio, 2014; Moosavi-Dezfooli et al., 2016) have shown the effectiveness of adversarial examples in misleading DNN-based systems; corrupting their integrity. They show how malicious inputs can be added to the training data to improve the resilience of the models. The erroneous behavior of a model can also be used to understand the root cause of some security vulnerabilities and generate countermeasures as shown in Moosavi-Dezfooli et al. (2016). **Mutation Testing: Alteration of ML model internals** To help protect DNNs against adversarial attacks, Wang et al. (2018) leveraged model mutation testing to detect adversarial examples at runtime. Using the following mutation operators, i.e., Gaussian Fuzzing (GF), Weight Shuffling (WS), Neuron Switch (NS), and Neuron Activation Inverse (NAI), they randomly mutated DNNs and compute the label change ratio (LCR) of both adversarial data and genuine data. They observed that adversarial examples have significantly higher label change ratio (LCR) under model mutation than original examples. Leveraging this observation, they proposed to use LCR to decide at runtime whether an input is adversarial or genuine. Model mutation testing can also be used to assess the quality of adversarial synthetic data and generate adversarial examples that are more subtle and effective for adversarial training. Conceptually, ML models identify high-level abstract features and patterns in the data and encode semantic information about how these set of features

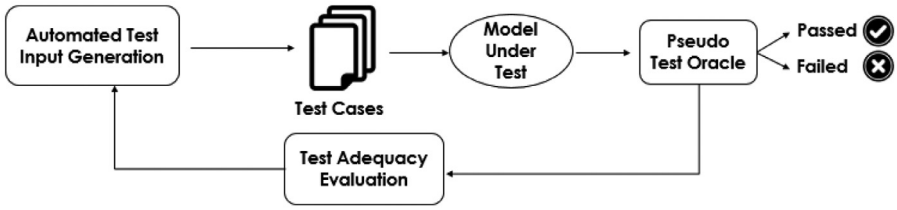

Fig. 4. Major components of a white-Box ML testing approach.

or patterns are related to the target outcome. In practice, ML models are exposed to multiple potential issues that can prevent them from learning the optimal mapping function, including inappropriate configuration (e.g., poor regularization) and implementation errors in the ML algorithm. It is challenging to identify the resulting incorrect behaviors of the model because the input space is large and manually labeled test data can only cover a small fraction of this space, leaving many corner-cases untested. Inspired by adversarial evasion attacks, researchers have started proposing testing approaches for ML models based on automated partial oracles that are inferred from applying data transformations, including imperceptible perturbations and affine transformations, on an initial human created oracle. Thanks to these automated partial oracles, it is now possible to generate large sets of test inputs automatically (Pei et al., 2017; Guo et al., 2018; Tian et al., 2018).

One major limitation of these black-box testing techniques is the representativeness of the generated adversarial examples. In fact, many adversarial models that generate synthetic images often apply only tiny, undetectable, and imperceptible perturbations, since any visible change would require manual inspection to ensure the correctness of the model's decision. This can result in strange aberrations or simplified representations in synthetic datasets, which in turn can have a hidden knock-on effects on the performance of a ML model when unleashed in a real-world setting. These black-box testing techniques that rely only on adversarial data (ignoring the internal implementation details of the models under test) often fail to uncover different erroneous behaviors of the model, even after performing a large number of tests. This is because the generated adversarial data often fails to cover the possible behaviors of the model adequately. This is an outcome that is not surprising given that the adversarial data is generated without considering information about the structure of the models. To help improve over these limitations, ML researchers have developed the white-box techniques described below, which use internal structure specificities to guide the generation of more relevant test cases.

5.1.2. White-box testing approaches for ML models

Recently, software researchers have started adapting concepts and systematic approaches based on well-known designs and practices from white-box software testing domain. As can be seen in the Fig. 4, these advanced testing approaches rely on three main components to improve the trustworthiness of DL systems:

1. Pseudo-Test Oracle, which allows identifying failed tests by distinguishing between the correct and incorrect obtained behaviors of the model under test.
2. Test Adequacy Evaluation, which allows estimating the fault-revealing ability of a given test suite to measure the degree/confidence that the test suite is adequate to terminate the testing process as test quality measurement.
3. Automated Test Input Generation, which allows exploring the input space and producing automatically effective test cases through maximizing one or multiple adopted test adequacy criteria.

In the following, we detail some examples of instances for each component of a white-box testing approach that have been proposed by relevant research works in the literature. **Pseudo-Test Oracle** Differential Testing Pei et al. proposed DeepXplore Pei et al. (2017), the first white-box approach for systematically testing deep learning models. DeepXplore is capable of automatically identifying erroneous behaviors in deep learning models without the need for manual labelling. Indeed, the approach circumvents the lack of a reference oracle, by using differential testing. Differential testing is a pseudo-oracle testing approach that has been successfully applied to traditional software that does not have a reference test oracle (McKeeman, 1998). It is based on the intuition that any divergence between programs' behaviors, solving the same problem, on the same input data is probably due to an error. Therefore, the test process consists of writing at first multiple independent programs to fulfill the same specification. Then, the same inputs are provided to these similar programs, which are considered as cross-referencing oracles. Differences in their executions are inspected to identify potential errors. In a same way, DeepXplore leverages a group of similar deep neural networks that solve the same problem to detect difference-inducing test cases that can represent erroneous and corner-cases behaviors under certain validity circumstances for input data validity. **Metamorphic Testing** Indeed, the differential testing is considered as costly pseudo-oracle, which requires the implementation of multiple similar models and also needs to run them all on each test input. That is why, the following white-box methods rely on metamorphic testing to compensate for the lack of a reference oracle. The latter is another pseudo-oracle software testing technique that allows identifying erroneous behaviors by detecting violations of domain-specific metamorphic relations (MR) (Chen et al., 1998). These MRs are defined across outputs from multiple executions of the test program with different inputs. The application of such test consists of performing some input changes in a certain way that allows testers to predict the output based on identified MRs, so any significant differences in output would break the relation, which would indicate the existence of errors in the program. Thus, as most of the models under test solve classification problems, researchers adopt the same rule of adversarial machine learning that constraints the generation of synthetic inputs to be close enough, i.e. semantically equivalent, to the genuine ones aiming at being able to attribute the same label to them. Exceptionally, for the regression problem such as predicting the steering angle in autonomous driving system, Tian et al. (2018) defined softer metamorphic relations between the steering angles of both original and derived driving scenarios. Briefly, they assume that the predicted angle for a transformed scene driving is correct if it varies from its genuine one to less than λ times the mean squared error produced by the original data set. λ is a configurable parameter that helps to strike a balance between the false positives and false negatives. **Test Adequacy CriteriaNeuronal Coverage** The first white-box test adequacy criterion adapted for DL models was *Neuron Coverage (NC)*, which estimates the amount of the neural network's logic explored by a set of inputs. This neuron coverage metric computes the rate of activated neurons in the neural network.

It was inspired by the code coverage metrics used for traditional software systems. Building on the pioneer work of Pei et al., Tian et al. proposed DeepTest (Tian et al., 2018), a tool for automated testing of DNN-based autonomous cars. In DeepTest, Tian et al. expanded the notion of neuron coverage proposed by Pei et al. for CNNs (Convolutional Neural Networks), to other types of neural networks, including RNNs (Recurrent Neural Networks). In addition, Guo et al. (2018) expanded DeepXplore Neuron's coverage measure with additional neuron selection strategies such as the selection of neurons that are frequently activated during model executions, and the prioritization of neurons with top high-value weights. The assumption being that these neurons may have a bigger influence on the model's behavior. Furthermore, Ben Braiek and Khomh (2019) defined two levels of neuron coverage and used it to build a novel test adequacy measure that is robust against plateauing during the optimization process (i.e., reaching a state of little or no change after a time of progress). In fact, the coverage-based measure captures both local-neuron coverage (i.e., neurons covered by a generated test input that were not covered by its corresponding original input) and global-neuron coverage (i.e., neurons covered by a generated test input that were not covered by all previous test inputs).

Indeed, these NC-based criteria allow assessing the diversity of DNN neuronal behavior triggered by testing inputs. This can be used to guide the generation of synthetic data that exhibit novel neuron state and enhance the diversity of inputs. Evaluations done by those research works (Pei et al., 2017; Tian et al., 2018; Ben Braiek and Khomh, 2019) have shown that the proposed DNN coverage criteria are correlated with the adversarial examples, which indicates that the unfamiliar inputs are more likely to trigger erroneous behaviors. Therefore, increasing the neuronal coverage promotes the diversity of the generated inputs, which likely results in more effective test cases. **MC/DC Coverage** Next, Sun et al. (2018a) examined the effectiveness of the neuron coverage metric introduced by DeepXplore and report that 100% neuron coverage can be easily achieved by a few test data points while missing multiple incorrect behaviors of the model. To illustrate this fact, they showed how 25 randomly selected images from the MNIST test set yield a close to 100% neuron coverage for an MNIST classifier. Thereby, they argue that testing DNNs should take into account the semantic relationships between neurons in adjacent layers in the sense that deeper layers use previous neurons' information represented by computed features and summarize them in more complex features. To propose a solution to this problem, they adapted the concept of Modified Condition/Decision Coverage (MC/DC) (Hayhurst, 2001) developed by NASA. The concepts of "decision" and "condition" in the context of DNN-based systems correspond to testing the effects of first extracted less complex features, which can be seen as potential factors, on more complex features which are intermediate decisions. Consequently, they specify each neuron in a given layer as a decision and its conditions are its connected input neurons from the previous layer. They propose a testing adequacy evaluation that is based on a set of four criteria inspired by MC/DC. As an illustration of proposed criteria, we detail their notion of *Sign-Sign(SS)* coverage, which is very close to the idea of MC/DC. Since the neurons' computed outputs are numeric continuous values, the *SS* coverage cannot catch all the interactions between neurons in successive layers. Since the changes observed on a neuron's output can be either a sign change or a value change, they added three additional coverage criteria to overcome the limitations of *SS, i.e.,* Value-Sign Coverage, Sign-Value Coverage, and Value-Value Coverage. These three additional criteria allow detecting different ways in which changes in the conditions can affect the models' decision. **Combinatorial Testing Coverage Criteria** Despite the relative success of neuronal coverage in estimating the behavioral changing of neural networks' activations, Ma et al. (2018c) remarked that the real DNN state space is very large and it can be relevant to take into account the interactions between the different neurons, however, given the size of neurons, this can lead to a combinatorial explosion. To help address this issue, they proposed DeepCT, which is an adaptation of combinatorial testing (CT) techniques to deep learning models, in order to reduce the testing coverage space. CT (Nie and Leung, 2011) has been successfully applied to test traditional software requiring many configurable parameters. It helps to sample test input parameters from a huge original space that are likely related to undetected errors in a program. For example, the t-way combinatorial test set covers all the interactions involving t input parameters, in a way that exposes efficiently the faults under the assumption of a proper input parameters' modeling. In DeepCT, K-way CT is adapted to allow for effectively selecting samples of neuron interactions inside different layers with the aim of decreasing the number of test cases. **Multi-granularity Coverage Levels**Ma et al. (2018a) generalized the concept of *neuron coverage* by proposing DeepGauge, a set of multi-granularity testing criteria for deep learning systems. DeepGauge measures the testing quality of test data (whether it being genuine or synthetic) in terms of its capacity to trigger both major function regions as well as the corner-case regions of DNNs (Deep Neural Networks). It separates DNN test coverage into two different levels.

At the neuron-level, the first criterion is a k-multisection neuron coverage, where the range of values observed during training sessions for each neuron are divided into k sections to assess the relative frequency of returning a value belonging to each section. In addition, the authors insist on the need for test inputs that are different enough from training data distribution to cover rare neurons' outputs. They introduced the concept of neuron boundary coverage to measure how well the test datasets can push activation values to go above and below a pre-defined bound (*i.e.,* covering the upper boundary and the lower boundary values). Their design intentions are complementary to Pei et al. in the sense that the k-multisection neuron coverage could potentially help cover the main functionalities provided by DNN. However, the neuron boundary coverage could relatively approximate corner-cases DNN's behaviors. The neuron-level coverage criteria are computed as follows : (1) *K-multisection Neuron Coverage (KMNC)*: the ratio of covered k-multisections of neurons; (2)*Neuron Boundary Coverage (NBC)*: the ratio of covered boundary region of neurons; (3)*Strong Neuron Activation Coverage (SNAC)*: the ratio of covered hyperactive boundary region.

At the layer-level, the authors leveraged recent findings (Kim et al., 2017; Zhou et al., 2018) that empirically showed the potential usefulness of patterns that are discovered within a regions of hyperactive neurons. These hyperactive neurons are often activated, *i.e.,* they render activation values relatively larger than the other neurons of the same layer, for example, one can compute the normalized activation value for each neuron, then, he selects the neurons yielding outputs higher than a predefined threshold (typically 0.2 Tian et al. (2018)). Through this process, we can identify the most important neurons of each layer. On the one hand, each layer allows DNN to characterize and identify particular features from input data and its main function is in large part supported by its top active neurons. Therefore, regarding the effectiveness in discovering issues, test cases should go beyond these identified hyperactive neurons in each layer. On the other hand, DNN provide the predicted output based on patterns recognized from a sequence features, including simple and complex ones. These features are computed by passing the summary information through hidden layers. Thereby, the combinations of top hyperactive neurons from different layers characterize the behaviors of DNN and the functional scenarios covered. Intuitively, test data sets should trigger other patterns of activated neurons in order to discover corner-

cases behaviors. The layer-level coverage criteria are computed as follows : (1)*Top-k Neuron Coverage (TKNC)*: the ratio of neurons in top-k hyperactived state on each layer; (2)*Bottom-k Neuron Coverage (BKNC)*: the ratio of neurons in top-k hypoactived state on each layer. In their empirical evaluation of DeepGauge, Ma et al. show that DeepGauge scales well to practical sized DNN models (*e.g.,* VGG-19, ResNet-50) and that it could capture erroneous behavior introduced by four state-of-the-art adversarial data generation algorithms (*i.e.,* Fast Gradient Sign Method (FGSM) (Goodfellow et al., 0000), Basic Iterative Method (BIM) (Kurakin et al., 2016), Jacobian-based Saliency Map Attack (JSMA) (Papernot et al., 2016), and Carlini/Wagner attack (CW) (Gu and Rigazio, 2014)). Therefore, a higher coverage of their criteria potentially plays a substantial role in improving the detection of errors in the DNNs. These positive results show the possibility to leverage this multi-level coverage criteria to create automated white-box testing frameworks for neural networks.

Surprise Adequacy Recently, Kim et al. (2018) proposed a fine-grained test adequacy metric, named Surprise Adequacy (SA) that quantifies how much surprising a given input is to the DNN under test with respect to the training data. The intuition behind this criterion is that effective test inputs should be sufficiently surprising compared to the training data. The surprise of an input is quantitatively measured as behavioural differences observed in a given input relatively to the training data. Kim et al. defined two concrete instances of their SA metric, given DNN's activations trace (AT) that represent a vector of neurons' activations : (1) Likelihood-based SA which uses Kernel Density Estimation (KDE) to estimate the probability density of each activation in AT, and computes the relative likelihood of new input's activation values with respect to estimated densities; and (2) Distance-based SA which uses the Euclidean distance between a given input's AT and the nearest AT of training data in the same class. Kim et al. evaluated their SA metrics using state-of-the-art adversarial generators (i.e., FGSM, BIM, JSMA, or CW) and different automated input generators such as DeepXplore gradient-based generator and DeepTest combined transformation generator, which are detailed more in the following paragraph. The authors report that SA can successfully capture the relative surprise of synthetic inputs. **Automated Test Input GeneratorsGradient-Based Optimization** In DeepXplore, the first white-box DNN testing framework, Pei et al. implement a wide variety of inputs' perturbations that induce visible differences(*e.g.,* different lighting, occlusion, etc.). Then, they formulate the objective of finding a large number of difference-inducing inputs while maximizing neuron coverage as a joint optimization problem. Thus, DeepXplore performs gradient ascent to solve efficiently this optimization problem using the gradient of the deep neural network with respect to the input. Its objective is to generate test data that provokes a different behavior from the group of similar deep neural networks under test in order to ensure a high neuronal coverage. We noticed that domain-specific constraints are added to generate data that is valid and realistic. In the end of the testing process, the generated data are kept for future training, to have more robustness in the model.

Coverage-guided Greedy Searching. Moreover, instead of injecting simple perturbations in input data, DeepTest focuses on generating realistic synthetic images by applying realistic image transformations like changing brightness, contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect, etc. They also mimic different real-world phenomena like camera lens distortions, object movements, different weather conditions, etc. They argue that generating inputs that maximize neuron coverage cannot test the robustness of trained DNN unless the inputs are likely to appear in the real-world. They provide a neuron-coverage-guided greedy search technique for efficiently finding sophisticated synthetic tests which capture different realistic image transformations that can increase neuron coverage in a self-driving car's DNNs.

GAN-Based Generation. DeepRoad (Zhang et al., 2018a) continued the same line of work as DeepTest, designing a systematic mechanism for the automatic generation of test cases for DNNs used in autonomous driving cars. Data sets capturing complex real-world driving situations are generated and Metamorphic Testing is applied to map each data point into the predicted continuous output. However, DeepRoad differs from DeepTest in the approach used to generate new test images. DeepRoad relies on a Generative Adversarial Network (GAN)-based method to provide realistic snowy and rainy scenes, which can hardly be distinguished from original scenes and cannot be generated by DeepTest using simple affine transformations. Zhang et al. argue that DeepTest synthetic image transformations, such as adding blurring/fog/rain effect filters, cannot simulate complex weather conditions. They claim that DeepTest's produced road scenes may be unrealistic, because simply adding a group of lines over the original images cannot reflect the rain condition or mixing the original scene with the scrambled "smoke" effect does not simulate the fog. To solve this lack of realism in generated data, DeepRoad leveraged a recent unsupervised DNN-based method (*i.e.,* UNIT) which is based on GANs and VAEs, to perform image-to-image transformations. UNIT can project images from two different domains (*e.g.,* a dry driving scene and a snowy driving scene) into a shared latent space, allowing the generative model to derive the artificial image (*e.g.,* the snowy driving scene) from the original image (*e.g.,* the dry driving scene). Evaluation results show that the generative model used by DeepRoad successfully generates realistic scenes, allowing for the detection of thousands of behavioral inconsistencies in well-known autonomous driving systems.

Symbolic Execution-Based Approach. Gopinath et al. (2019) introduced DeepCheck, a lightweight symbolic-execution based approach for testing DL models, that transforms a DNN into a program by translating the activations into IF-Else branch structures, to create the translated program paths. DeepCheck is capable of performing symbolic analysis on a translated DNN program to solve two challenging problems (1) identification of important pixels (for attribution and adversarial generation); and (2) creation of 1-pixel and 2-pixel attacks. Experimental results using the MNIST data-set show that DeepCheck's lightweight symbolic analysis can effectively find adversarial examples resulting from the perturbation of the most valuable pixels or pixel-pairs for the classification of the corresponding modified images.

Sun et al. (2018b) applied concolic testing (Sen et al., 2005) to DNNs. Concolic testing combines concrete executions and symbolic analysis to explore the execution paths of a program that are hard to cover by blind test case generation techniques such as random testing. The proposed adaptation of concolic testing to DNNs leverages state-of-the-art coverage criteria and search approaches. The authors first formulate an objective function that contains a set of existing DNN-related coverage requirements using Quantified Linear Arithmetic over Rationals (QLAR). Then, their proposed method incrementally finds inputs data that satisfy each test coverage requirement in the objective. Iterating over the set of requirements, the concolic testing algorithm finds the existing test input that is the closest data point to satisfy the current requirement following an evaluation based on concrete execution. Relying on this found instance, it applies symbolic execution to generate a new data point that satisfies the requirement and adds it to the test suite. The process finishes by providing a test suite that helps reach a satisfactory level of coverage. To assess the effectiveness of their proposed approach, they evaluated the number of adversarial examples that could be detected by the produced test suites. **Constraint Programming** Regarding the combinatorial testing coverage criteria, constraint-based solvers (*i.e.,* by linear pro-

gramming using the CPLEX solver (IBM, 2018)) are able to generate some DNN-related K-way coverage criteria. Then, given the initial test data sets, it produces new test data by perturbing the original data within a prefixed value range, while ensuring that previously generated CT coverage criteria are satisfied on each layer. Ma et al. (2018c) conducted an empirical study of their CT framework, DeepCT, comparing the 2-way CT cases with random testing in terms of the number of adversarial examples detected. They observed that random testing was ineffective even when a large number of tests were generated. In comparison, DeepCT performed well even when only the first several layers of the DNN were analyzed, which shows some usefulness for their proposed CT coverage criteria in the context of adversarial examples detection and local-robustness testing. However, even though solvers like CPLEX represent the state-of-the-practice, their scalability remains an issue. Hence, the effectiveness of the proposed DeepCT approach on real-world problems using large and complex neural networks remains to be seen. **Coverage-Guided Fuzzing** Different from the previous works, Xie et al. Xie et al. (2018) proposed, DeepHunter, a scalable and general-purpose coverage-guided fuzz testing framework for DNN software. It performs semantically-preserving image mutations to generate new synthetic inputs from original ones including pixel-value mutations (contrast, blur, etc.) and affine transformations (rotation, translation, etc.). Then, it runs a test suite as a batch of input tests and prioritizes the tests selection using the multi-level coverage criteria proposed in DeepGauge. Indeed, the batch prioritization gives high chances to the generated batches that have triggered new uncovered regions; so it guides the test generation towards enhancing the chances of satisfying the coverage criteria. The effectiveness of DeepHunter was investigated on 3 popular datasets (MNIST, CIFAR-10, ImageNet) and 7 DNNs with diverse complexities and it has successfully demonstrated its usefulness in facilitating defect detection and model quality evaluation.

Differential Fuzzing. Guo et al. (2018) observed that traditional coverage-guided fuzzing techniques (which consist in randomly mutating input data to generate new inputs that may increase the coverage criteria and eventually help uncover erroneous model behavior) often fail to find relevant input data because of the huge size of the input data space. Finding corner-cases regions through a blind mutation of an input corpus is hard and expensive when the input data space is huge. To solve this problem, they proposed DL Fuzz, a differential fuzzing testing framework where the maximization of neuron coverage and prediction difference between original and mutated inputs is formulated as a joint optimization problem, that can be solved efficiently using gradient-based ascent optimizers. Starting with a random list of data inputs, DL Fuzz iteratively computes the gradient of the optimization objective with respect to each input data and applies it as perturbations to the input data in order to obtain new mutated test input. Then, the $L2$ distance between the original input and the mutated input is evaluated to guarantee that the performed perturbation is invisible to humans and that the two inputs share the same prediction output. Finally, all the generated inputs that contributed to increase the neuron coverage are kept in a maintained input data corpus. **Search-Based Approach** Existing automated input generators have important limitations: (1) there is a lack of variability on input transformations when using gradient-based optimizers. For example, in computer vision applications, gradient-based transformations can perform imperceptible pixels' perturbations, but they cannot infer optimal parameters for affine transformations such as rotation or translation. (2) The second limitation is blindness of coverage-guided fuzzing that do not guarantee reaching the objective, when it relies on simple strategies without any optimization routines. Recently, Ben Braiek and Khomh (2019) proposed, DeepEvolution, the first search-based testing method specialized for DNN software that relies on population-based metaheuristics to explore the search space of semantically-preserving metamorphic transformations. Since these metaheuristics are gradient-free optimizers, they ensure the maximization of neuron coverage-based fitness, while keeping a wide variety of input transformations thanks to their flexibility (as they do not require prior assumptions). The evaluation of DeepEvolution on computer-vision DL models showed that it succeeds in boosting the neuronal coverage of DNNs under test and successfully exposes corner-cases behaviors on popular image recognition models trained on both MNIST and CIFAR-10 datasets.

5.2. Approaches that aim to detect errors in ML code implementations

Given the stochastic nature of most ML algorithms and the absence of oracles, most existing testing techniques are inadequate for ML code implementations. As a consequence, the ML community have resorted to numerical testing, property-based testing, metamorphic testing, mutation testing, coverage-guided fuzzing testing, and proof-based testing techniques to detect issues in ML code implementations. In the following, we present the most prominent techniques.

5.2.1. Numerical-based testing: Finite-difference techniques

Most machine learning algorithms are formulated as optimization problems that can be solved using gradient-based optimizers, such as gradient descent or L-BFGS (*i.e.,* Limited-memory BroydenFletcherGoldfarbShanno algorithm). The correctness of the objective function gradients that are computed with respect to the model parameters, is crucial. In practice, developers often check the accuracy of gradients using a finite difference technique that simply consists of performing the comparison between the analytic gradient and the numerical gradient. The analytic gradient can be either manually inferred and hard-coded by the developer or automatically generated by the automatic differentiation component of a DL library. The numerical gradient can be estimated using finite difference approximations. Nevertheless, because of the increasing complexity of models' architectures, this technique is prone to errors. To help improve this situation, Karpathy Karpathy, 2018 have proposed a set of heuristics to help detect faulty gradients. In the following, we elaborate on each of these heuristics.

a)Use of the centered formula Instead of relying on the traditional gradient formula, Karpathy recommends using the centered formula from Eq. (1), which is more precise. The Taylor expansion of the numerator indicates that the centered formula has an error in the order of $O(h^2)$, while the standard formula has an error of $O(h)$.

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \tag{1}$$

b) Use of relative error for the comparison As mentioned above, developers perform gradient checking by computing the difference between the numerical gradient f'_n and the analytic gradient f'_a. This difference can be seen as an absolute error and the aim of the gradient checking test is to ensure that it remains below a pre-defined fixed threshold. With deep neural networks, it can be hard to fix a common threshold in advance for the absolute error. Karpathy recommends fixing a threshold for relative errors. So, for a deep neural network's loss that is a composition of ten functions, a relative error 2 of $1\,\exp^{-2}$ might be acceptable because the errors build up through backpropagation. Conversely, an error of $1\,\exp^{-2}$ for a single differentiable function likely indicates an incorrect gradient.

$$\frac{|f'_a - f'_n|}{max(|f'_a|, |f'_n|)} \tag{2}$$

c) Use of double precision floating point Karpathy recommends avoiding using single precision floating point to perform gradient

checks, because it often causes high relative errors even with a correct gradient implementation.

d) Stick around active range of floating point To train complex statistical models, one needs large amounts of data. So, it is common to opt for mini-batch stochastic gradient descent and to normalize the loss function over the batch. However, if the back-propagated gradient is very small, additional divisions by data inputs count will yield extremely smaller values, which in turn can lead to numerical issues. As a solution to this issue, Karpathy recommends computing the difference between values with minimal magnitude, otherwise one should scale the loss function up by a constant to bring the loss to a denser floats range, ideally on the order of 1.0 (where the float exponent is 0).

e) Use only a few random data inputs The use of fewer data inputs reduces the likelihood to cross kinks when performing the finite-difference approximation. These kinks refer to non-differentiable parts of an objective function and can cause inaccuracies in the numerical gradients. For example, let's consider an activation function *ReLU* that has an analytic gradient at the zero point, *i.e.,* it is exactly zero. The numerical gradient can compute a non-zero gradient because it might cross over the kink and introduce a non-zero contribution. Karpathy strongly recommends performing the gradient checking for a small sample of the data and inferring the correctness of gradient for an entire batch, because it makes this sanity-check fast and more efficient in practice.

f) Check only a few dimensions Recent statistical models are more and more complex and may contain thousands parameters with millions of dimensions. The gradients are also multi-dimensional. To mitigate errors, Karpathy recommends checking a random sample of the dimensions of the gradient for each separate model's parameter, while assuming the correctness of the remaining ones.

g) Turn off dropout/regularization penalty Developers should be aware of the risk that the regularization loss overwhelms the data loss and masks the fact that there exists a large error between the analytic gradient of data loss function and its numerical one. Such circumstance can result in a failure to detect an incorrect implementation of the loss function or its corresponding gradient using the finite-difference approximation technique. Karpathy recommends turning off regularization and checking the data loss alone, and then the regularization term, independently. Moreover, recent regularization techniques applied to deep neural networks such as dropout, induce a non-deterministic effect when performing gradient check. Thereby, to avoid errors when estimating the numerical gradient, it is recommended to turn them off. An alternative consists in forcing a particular random seed when evaluating both gradients.

5.2.2. Property-based testing

Property-based testing is a technique that consists in inferring the properties of a computation using the theory and formulating invariants that should be satisfied by the code. Using the formulated invariants, test cases are generated and executed repeatedly throughout the computation to detect potential errors in the code. Using property-based testing, one can ensure that probability laws hold throughout the execution of a model. For example, one can test that all the computed probability values are non-negatives. For a discrete probability distribution, as in the case of a classifier, one can verify that the probabilities of all events add up to one. Also, marginalization can be applied to test probabilistic classifiers.

Roger et al. Grosse and Duvenaud (2014) applied property-based testing to detect errors in implementations of MCMC (Markov chain Monte Carlo) samplers. They derived by hand the update rules for individual variables using the theory, *i.e.,* they sampled a random variable from its conditional distribution. Then, they wrote a single iterative routine including those rules. To test the correctness of the produced samples, they verified that the conditional distribution was consistent with the joint distribution at each update iteration.

Karpathy, 2018 recommend the verification of the following properties.

- Initial random loss: when training a neural network classifier, turn off the regularization by setting its corresponding strength hyperparameter to zero and verify that initial softmax loss is equal $-\log\left(\frac{1}{N_c}\right)$ with N_c: number of label classes. One expect a diffuse probability of $\frac{1}{N_c}$ for each class.
- Overfitting a tiny dataset: Keeping the regularization term turned off, extract a sample portion of data, (one or two examples inputs from each class) in order to ensure that the training algorithm can achieve efficiently zero loss. Breck et al. also recommend watching the internal state of the model on small amounts of data with the aim of detecting issues like numerical instability that can induce invalid numeric values like NaNs or infinities.
- Regularization role: increase the regularization strength and check if the data loss is also increasing.

Current properties to test DNN models are too coarse-grained and allow DL engineers to only validate if the training program is able to minimize the loss and produce a functioning model. To complete these properties by fine-grained ones, Braiek and Khomh (2019) proposed TFCheck, a property-based testing framework for debugging DNN training programs. TFCheck implements a set of verification routines from the literature and uses them to continuously check a DL training program, to uncover potential training issues such as saturated neurons' activations or highly-fluctuating loss. To assess the effectiveness of TFCheck at detecting DNN training issues automatically, Braiek and Khomh conducted a case study using real-world, mutants, and synthetic training programs. The results of this case study show that using TFCheck, DL engineers can successfully detect training issues in a DNN code implementations.

Another testing technique that shares the same philosophy as property-based testing is metamorphic testing.

5.2.3. Metamorphic testing

Murphy et al., 2008 introduced metamorphic testing to ML in 2008. They defined several Metamorphic Relationships (MRs) that can be classified into six categories (*i.e.,* additive, multiplicative, permutative, invertive, inclusive, and exclusive). The performed transformations include adding a constant value to numerical attributes; multiplying numerical attributes by a constant value; permuting the order of inputs; reversing the order of inputs; removing a portion of inputs; adding additional instances. In fact, the defined MRs are quite generic and can be applied for different types of machine learning algorithms (ranking, supervised classifiers, unsupervised clustering, etc...). Murphy et al., 2007 manually assessed the effectiveness of their defined MRs on three well-known ML applications: Marti-Rank, SVM-Light (Support Vector Machine with a linear kernel), and PAYL (Murphy et al., 2007), and concluded that they can be used to test ML applications efficiently. Xie et al. (2011a) proposed MRs specialized for testing the implementations of supervised classifiers. The MRs are based on five types of transformations: (1) application of affine transformations to input features;(2) permutation of the order of labels or features; (3) addition of uninformative and informative new features; (4) duplication of some training instances; and (5) removal of arbitrary classes or instances. The evaluation of these new MRs was conducted on the implementation of *k*-Nearest Neighbors (kNN) and Naive Bayesian (NB) from Weka Hall et al. (2009). Using the MRs, the authors were able to uncover defects in the implementation

of NB provided by Weka. In 2011, Xie et al. Xie et al. (2011b) further evaluated their MRs using mutation testing. They found that their proposed MRs were able to reveal 90% of the injected faults in Weka (injected by MuJava Ma et al. (2005)).

Recent research works Dwarakanath et al. (2018) has investigated the application of metamorphic testing to more complex machine learning algorithms such as SVM with non-linear kernel and deep residual neural networks (ResNET). For SVM, they applied transformations such as: changing features or instances orders, linear scaling of the features. For deep learning models, since the features are not directly available, they proposed to normalise or scale the test data, or to change the convolution operations order. They used the Python code mutation tool MutPy, to mutate the training program and simulate implementation faults. Their MRs were able to find 71% of the injected faults.

5.2.4. Mutation testing

Ma et al. (2018b) proposed DeepMutation, which adapts mutation testing (Jia and Harman, 2011) to DNN-based systems with the aim of evaluating the test data quality in terms of its capacity to detect faults in the programs. To build DeepMutation, Ma et al. defined a set of source-level mutation operators to mutate the source of a ML program by injecting faults. These operators allow injecting faults in the training data (using data mutation operators) and the model training source code (using program mutation operators). After the faults are injected, the ML program under test is executed, using the mutated training data or code, to produce the resulting mutated DNNs. The data mutation operators are intended to introduce potential data-related faults that could occur during data engineering (*i.e.,* during data collection, data cleaning, and/or data transformation). Program mutation operators' mimic implementation faults that could potentially exist in the model implementation code. These mutation operators are sematic-based and specialized for DNNs' code. Training models from scratch following source-level mutations is very time-consuming since advanced deep learning systems require often tens of hours and even days to learn the model's parameters. Moreover, manually designing specific mutation operators using information about faults occurring in real-world DNNs systems is challenging, since it is difficult to imagine and simulate all possible faults that occur in real-world DNNs. To circumvent the cost of multiple re-execution and fill in the gap between real-world erroneous models and mutated models, the authors define model-level mutation operators to complement source-level mutation operators. These operators directly change the structure and the parameters of neural network models to scale the number of resulted mutated models for testing ML programs in an effective way, and for covering more fine-grained model-level problems that might be missed by only mutating training data and/or programs. Once the mutated models are produced, the mutation testing framework assesses the effectiveness of test data and specify its weaknesses based on evaluation metrics related to the killed mutated models count. ML engineers can leverage this technique to improve data generation and increase the identification of corner-cases DNN behaviors.

5.2.5. Coverage-Guided fuzzing

Odena and Goodfellow (2018) developed a coverage-guided fuzzing framework specialized for testing neural networks. Coverage-guided fuzzing has been used in traditional software testing to find critical errors. For ML code, the fuzzing process consists of handling an input corpus that evolves through the execution of tests by applying random mutation operations on its contained data and keeping only interesting instances that allow triggering new program behavior. Iteratively, the framework samples an input instance from testing data corpus and mutates it in a way that constrains the difference between the mutated input and its original version to have a user-configurable L_∞ norm. This ensures that mutated instances remain associated with the same label as the original input. Then, it checks whether the corresponding state vector is meaningfully different from the previous ones using a fast approximate nearest neighbor algorithm based on a pre-specified distance. Each test input that is relatively far from the existing nearest neighbor is added to the set of test cases. The effectiveness of the proposed TensorFuzz framework was assessed using three known issues in neural networks' implementations. Results show that TensorFuzz surpasses random search in : (1) finding NaNs values in neural network models trained using numerical unstable cross-entropy loss, (2) uncovering divergences in decision between a model that is encoded with 32-bit floating point real and its quantized version that is encoded with only 16-bit, and (3) surfacing undesirable behavior in character level language RNN models.

5.2.6. Proof-based testing

Selsam et al. (2017) proposed to formally specify the computations of ML programs and construct formal proofs of written theorems that define what it means for the programs to be correct and error-free. Using the formal mathematical specification of a ML program and a machine-checkable proof of correctness representing a formal certificate that can be verified by a stand-alone machine without human intervention, ML developers can find errors in the code. Using their proposed approach, Selsam et al. analyzed a ML program designed for optimizing over stochastic computation graphs, using the interactive proof assistant *Lean*, and reported it to be bug-free. However, although this approach allows detection of errors in the mathematical formulation of the computations, it cannot help detect execution errors due to numerical instabilities, such as the replacement of real numbers by floating-point with limited precision.

6. The road ahead

Fig. 5 summarizes the literature on ML program testing along the key testing phases presented in Fig. 2.

As one can see, there have been some efforts to develop data cleaning and features processing verification tools (*i.e.,* Hynes et al., 2017; Krishnan et al., 2015, 2016, 2017). However, although these approaches can help detect errors in data pipelines, they do not support the end to end development and deployment process of ML programs. Specific tests should be written to ensure that the quality of data pipelines is not regressed over time due to unstable and epsilon features. The research community should consider devoting more effort to the development of efficient testing techniques that can help ML developers ensure quality in their feature engineering infrastructure. For example, unit tests should be developed to help verify that data invariants hold in new or augmented training datasets. Unit tests are also needed to test that each transformer operation included in the data pipeline performs as expected. Integration tests are also needed to validate that a pipeline is working correctly, *i.e.,* it is providing well-represented features that can feed the model efficiently. These integration tests can also help ensure that each newly selected feature contributes significantly to the model, and that none of the existing features have become obsolete.

Regarding the model, researchers have adapted traditional software testing methods to generate adversarial examples based on ML model's structure coverage criteria, *e.g.,* Nie and Leung (2011), Sun et al. (2018b), Ma et al. (2018a), Ma et al. (2018c), Moosavi-Dezfooli et al. (2016), and Sun et al. (2018a). More advanced approaches such as Pei et al. (2017), Tian et al. (2018), and Zhang et al. (2018a), propose input test data generators based on real-world phenomenon, in order to mimic realistic situations,

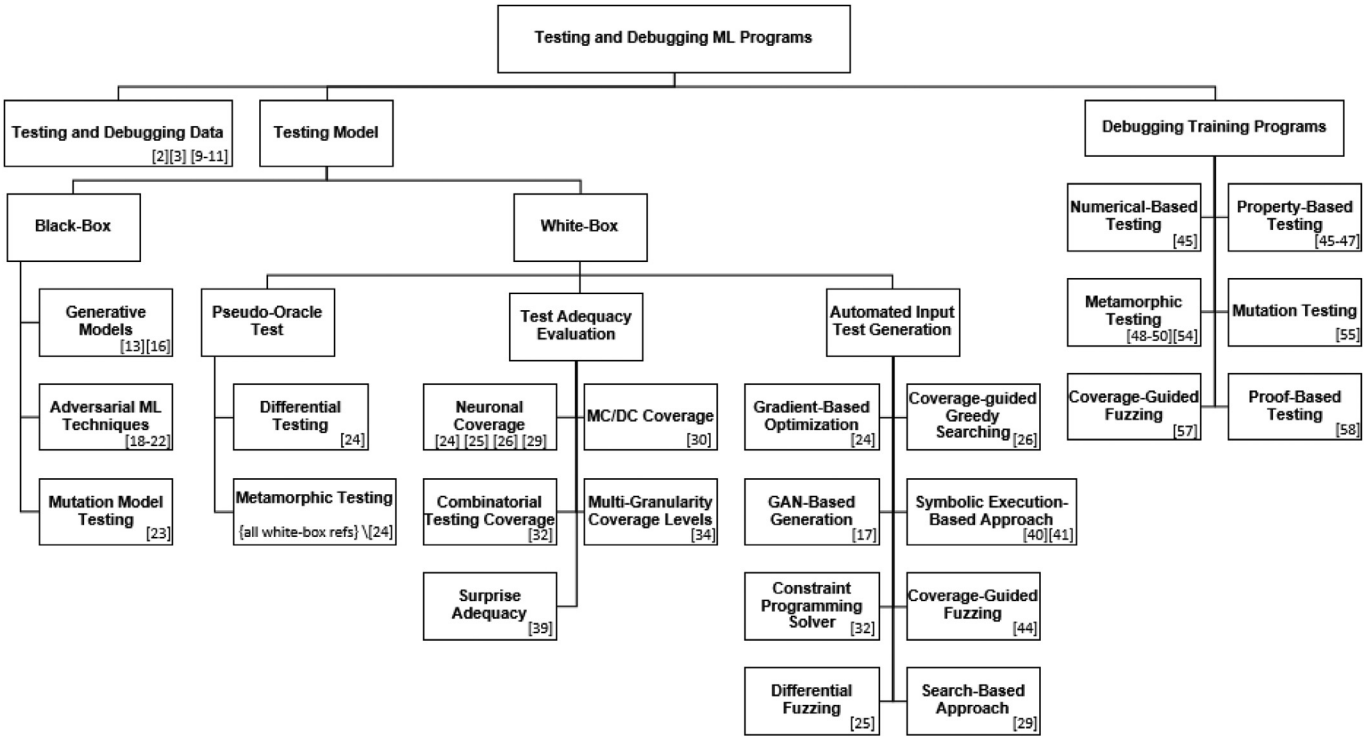

Fig. 5. Overview of literature on testing ML program components.

e.g., by changing an image input in a way to simulate the effects of camera lens distortions or lighting conditions or transforming a normal weather driving scene into a rainy or snowy driving scene. Although these approaches allow identifying not only model issues but also data issues (in terms of the lack of training data covering different real-world situations and contexts of use), they still miss many corner cases. More efficient automated corner cases data generators are needed. However, generating corner cases automatically is challenging given that it is impossible to enumerate all data points that are close to training data and which can potentially expose erroneous model's behaviors. Further works are also needed to propose more effective metrics of test coverage that reflect precisely the amount of logic and rules learned from data that are really involved in the testing. Moreover, collaborations with application domain experts are required to help prioritize the fundamental real-world situations that should be taught to the models. This can help build more reliable models that learn more robust invariants and high-level hidden patterns.

Concerning ML software implementations, the proposed proof-based testing approaches (Selsam et al., 2017) can exclusively detect potential mathematical design issues, since they help developers to specify requirements and mathematical specifications into formal proofs that can help ensure that the generated code has a correct design. A correct design does not guarantee a proper execution in floating-point computation on CPUs and GPUs. Moreover, these proof-based testing approaches do not help test the ML code written using programming languages such as python or C++. These proof-based approaches should be used in combination with tools such as TensorFuzz (Odena and Goodfellow, 2018) and other coverage-guided fuzzing tools that can identify numerical issues such as underflow and overflow problems and inconsistencies resulting from model quantization. These tools can cover not only execution environment numerical issues but also mathematical design issues, especially, the unstable mathematical functions that may return NaNs or Infs given particular inputs.

Regarding the mutation testing (Ma et al., 2018b), metamorphic testing (Dwarakanath et al., 2018; Xie et al., 2011; Murphy et al., 2007; Murphy et al., 2008), numerical-based testing (Karpathy, 2018), and property-based testing (Grosse and Duvenaud, 2014; Karpathy, 2018) approaches proposed in the literature to test ML programs. Although in theory they can help ML developers detect issues in both written code and mathematical design (since the buggy models would likely break some of the assumptions checked by these testing techniques), their effectiveness on real programs is still an open question. So far, these approaches have been evaluated mostly on mutants and/or few buggy synthetic code examples. More evaluations are needed to assess their effectiveness in testing real-world ML programs such as the TensorFlow programs examined by Zhang et al. (2018b). Results of such evaluations could help researchers further improve these testing approaches. To allow for a better testing of ML programs, we also recommend that the research community develops architectural methodologies to help ML developers architect their ML programs in a modular way that allow writing unit and integration tests effectively.

Very few research efforts (Sculley et al., 2015; Sculley et al., 2014) have discussed issues related to the deployment of ML models in production. Post-deployment testing and system performance monitoring are also critical aspects to be investigated. First, the data pipeline that actually extracts input features may differ for training and deployment environments. It is therefore important to verify key invariants on both training and deployment environments, and test that serving features compute the same expected values as training features. Second, it is important to test for numerical instabilities (*i.e.,* NaNs and infinities) on models in-production (*i.e.,* as a post-deployment test), because the execution environment is likely to be different from the training environment. Third, the system in-production should be monitored via specific regression testing, in order to quickly identify (i) any deterioration in terms of prediction quality on served data, and (ii) any degradation in terms of resources-consuming performance (such as

higher serving latency value or excessive RAM usage). These symptoms could indicate the occurrence of defects resulting from outside changes, whether in served data or infrastructure.

Machine learning technology is enabling software systems to make autonomous decisions. Nowadays, ML software systems are used to make decisions in credit evaluation (Olson, 2011), medical diagnosis (Strickland, 2016), and even criminal sentencing (Angwin et al., 2016). Since ML software applications are data-driven applications that rely heavily on training data, they can be affected by sample bias, measurement bias, or prejudice bias. ML models can also be subject to algorithm bias. Biased ML models are likely to make unfair decisions (i.e., decisions that disproportionately hurt (or benefit) people with certain sensitive attribute values, e.g., females, persons of colour). In fact, it has been found in 2016 that COMPAS, the algorithm used for recidivism prediction produces much higher false positive rate for people of colour than white people (Angwin et al., 2016). Recently, XING, a professional network platform similar to LinkedIn, was found to rank less qualified male candidates higher than more qualified female candidates (Lahoti et al., 2018). Therefore, the scope of testing ML-based software systems should be expanded beyond functional correctness and robustness and should include fairness and testing, to ensure the non-discrimination of decisions made by these intelligent systems. Recently, A metric-based fairness testing approach was proposed by Galhotra et al. (2017). Indeed, the approach rely on a set of causality-based measures of discrimination that measure the fraction of inputs for which changing specific characteristics causes the output to change. Using these metrics, they can generate automatically test suites allowing the detection of any sort of discrimination adopted by the under test ML models. The test cases are generated by modifying training data with respect to a sensitive attribute aimed at verifying if the modification causes a change in the outcome. The fairness test with respect to one sensitive attribute (SA) consists of validating that varying the SA value for any input does not alter the output. In the evaluation of the approach, Galhotra et al. investigated discriminations in software systems and found it to be prevalent, even when fairness is an explicit design goal, i.e., input features correlated to SAs have been removed (Calmon et al., 2017) or the optimization objective includes a regularization component to penalize the dependence on SA-related features (Zafar et al., 2017).

The proposed software testing method for ensuring the fairness of ML systems' decisions highlights the importance of running tests to validate the respect of fairness requirements in software systems. Nevertheless, there are main However, some important challenges remain: (1) existing causal discrimination mitigation techniques only support categorical variables for both inputs and outputs. More generic and accurate discrimination measures are needed to extend these proposed discrimination measures to a broader class of data types. (2) Performing optimization techniques to reduce the size of test inputs set is crucial to ensure the effectiveness of proposed fairness testing, especially when dealing with high-dimensional inputs. For example, combinatorial testing can be leveraged to minimize the generated tests, through identifying in prior that certain configuration of features' changes cannot affect a particular test's output. These configurations can be discarded to avoid useless test cases.

7. Related work

Software testing is a very active research field. There is an abundant body of literature about approaches for testing software programs. In the following, we discuss the most relevant works for testing ML programs and report about previous literature reviews on this topic.

On testing "Non-testable" Programs : Weyuker (1982) examined challenges and approaches for testing "non-testable" programs. A program is considered to be "non-testable" when there is no reference oracle (as it is the case for ML programs) or when it is practically very difficult to determine the correct answers. Weyuker identified three categories of "non-testable" programs: (1) programs that produce so many outputs that it is practically impossible to verify them all; (2) program for which developers and testers have different specifications and thus a tester misconception exists; (3) programs that were written to determine an answer (such as ML programs where developers aim to learn the application logic from data without any prior knowledge on it).

Weyuker (1982) also outlined strategies to circumvent the difficulty of testing "non-testable" programs. One of these strategies consist in assigning domain-specific measures of likelihood to different values in order to restrict the range of plausible results and exclude incorrect ones. For scientific software, one testing technique consists of using properties of the computation that are known from the theory, and repeatedly verifying that they remain invariant throughout the computations. Such verifications can help identify error bounds. Another testing approach identified for "non-testable" program is dual-coding, which is inspired from n-version programming. It consists in developing independently another program that fulfills the same specification as the program under test and comparing the results given by the two systems.

Challenges of Testing Machine Learning Applications : Huang et al. (2018) analyzed the main challenges of testing the ML models that perform a classification task. They report that an effective model testing approach should generate test cases that trigger model's incorrect behaviors similarly to traditional software testing where the effectiveness of test cases depends on the amount of defects that are detected. They focused on the model testing approaches that propose a technique to automate the generation of corner-cases (which are test cases that have a high chance of exposing an erroneous behavior of the model under test). These automated testing approaches are based on: (1) a corner cases test inputs generation algorithm and (2) an effective implementation of the algorithm to create a larger number of corner-cases test inputs automatically. Huang et al. classified the generation of test inputs techniques in three category: (1) gradient ascent algorithms that solve the generation of corner cases input as an optimization problem, (2) transformations based on application scenarios that mimic realistic conditions to provide several test inputs in different real-world situations, and (3) adversarial example algorithms that generate corner-cases inputs that are always slightly different from the source cases by adding an imperceptible non-random perturbation that can fool the model under test.

Software Quality Practices for Scientific Software : Kelly et al. (2008) conducted several interviews with scientists writing computational software. They aimed to understand software quality assurance practices in the community of developers of scientific software. They report that these developers mainly conduct model testing and rarely test the resulting software. The interviewed scientists and industrial researchers revealed that correctness is the prime quality attribute of interest for scientific software. Scientific developers focus on testing if the calculations expressed in the software code give answers that agreed sufficiently with expected answers. Quality attributes such as scalability, extensibility, or maintainability are rarely addressed. The authors concluded their study with practical guidelines for scientists and researchers interested in improving the quality of computational software systems.

Software Quality Practices for Machine Learning Applications : Masuda et al. (2018) examined the challenges of software quality assurance for ML-as-a-services (MLaas), which are ML services available through APIs on the cloud. They observed a lack

of rigourous evaluations of the quality of these systems. They also observed a lack of specialized techniques adapted to the specificity of these ML based systems. They also highlighted problems related to the deployment of ML models in real-world scenarios and discuss the possibility of adapting well-known software engineering methodologies for ML applications.

To the best of our knowledge, there is no previous work that review all existing testing practices for ML programs. We hope that our comprehensive review of software testing practices will help fill this gap in the literature, and help assist ML developers in identifying the right approach to improve the reliability of their ML-based systems.

8. Conclusion and future research

The inductive nature of ML programs makes it difficult to reason about their behavior. Recently, researchers have started to develop new testing techniques to help ML developers detect debug and test ML programs. In this paper, we describe the generic process of a ML program creation, from data preparation to the deployment of the model in production, and explain the main sources of faults in a ML program. Next, we review testing techniques proposed in the literature to help detect these faults both at the model and implementation levels; explaining the context in which they can be applied as well as their expected outcome. We also identify gaps in the literature related to the testing of ML programs and suggest future research directions for the scientific community. By consolidating the progress made in testing ML programs in a single document, we hope to equip ML and software engineering communities with a reference document on which future research efforts can be built. Practitioners can also use this paper to learn about existing testing techniques for ML programs, which in turn can help improve the quality of their ML programs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Houssem Ben Braiek: Investigation, Data curation, Conceptualization, Writing - original draft. **Foutse Khomh:** Methodology, Writing - review & editing, Supervision.

References

Angwin, J., Larson, J., Mattu, S., Kirchner, L., 2016. Machine bias. ProPublica, May 23.

Ben Braiek, H., Khomh, F., 2019. Deepevolution: a search-based testing approach for deep neural networks. In: 2019 IEEE International Conference on Software Maintenance and Evolution. IEEE.

Braiek, H.B., Khomh, F., 2019. Tfcheck: a tensorflow library for detecting training issues in neural network programs. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 426–433.

Calmon, F., Wei, D., Vinzamuri, B., Ramamurthy, K.N., Varshney, K.R., 2017. Optimized pre-processing for discrimination prevention. In: Advances in Neural Information Processing Systems, pp. 3992–4001.

Chen, T.Y., Cheung, S.C., Yiu, S.M., 1998. 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.

Doersch, C., 2016. Tutorial on variational autoencoders arXiv:1606.05908.

Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R.M., Bose, R., Dubash, N., Podder, S., 2018. 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. ACM, pp. 118–128.

Engstrom, L., Tsipras, D., Schmidt, L., Madry, A., 2017. A rotation and a translation suffice: fooling cnns with simple transformations arXiv:1712.02779.

Fischer, A., Igel, C., 2012. An introduction to restricted boltzmann machines. In: iberoamerican Congress on Pattern Recognition. Springer, pp. 14–36.

Galhotra, S., Brun, Y., Meliou, A., 2017. Fairness testing: testing software for discrimination. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp. 498–510.

Gibbs, S.. https://www.theguardian.com/technology/2018/may/08/ubers-self-driving-car-saw-the-pedestrian-but-didnt-swerve-report.

Gilmer, J., Adams, R.P., Goodfellow, I., Andersen, D., Dahl, G.E., 2018. Motivating the rules of the game for adversarial example research arXiv:1807.06732.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative adversarial nets. In: Advances in neural information processing systems, pp. 2672–2680.

Goodfellow, I. J., Shlens, J., Szegedy, C.,. Explaining and harnessing adversarial examples (2014). arXiv:1807.06732 1412.6572

Gopinath, D., Păsăreanu, C.S., Wang, K., Zhang, M., Khurshid, S., 2019. Symbolic execution for attribution and attack synthesis in neural networks. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. IEEE Press, pp. 282–283.

Grosse, R.B., Duvenaud, D.K., 2014. Testing MCMC code arXiv:1412.5218.

Gu, S., Rigazio, L., 2014. Towards deep neural network architectures robust to adversarial examples arXiv:1412.5068.

Guo, J., Jiang, Y., Zhao, Y., Chen, Q., Sun, J., 2018. 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. ACM, pp. 739–743.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. ACM SIGKDD explorations newsletter 11 (1), 10–18.

Hayhurst, K.J., 2001. A Practical Tutorial on Modified Condition/Decision Coverage. DIANE Publishing.

Hjelm, R.D., Jacob, A.P., Che, T., Trischler, A., Cho, K., Bengio, Y., 2017. Boundary-seeking generative adversarial networks arXiv:1702.08431.

Huang, S., Liu, E.-H., Hui, Z.-W., Tang, S.-Q., Zhang, S.-J., 2018. Challenges of testing machine learning applications.. Int. J. Perform. Eng. 14 (6).

Hynes, N., Sculley, D., Terry, M., 2017. The data linter: lightweight, automated sanity checking for ML data sets in NIPS MLSys. Workshop.

IBM. https://www.ibm.com/de-de/products/ilog-cplex-optimization-studio/details.

Jia, Y., Harman, M., 2011. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Jo, J., Bengio, Y., 2017. Measuring the tendency of CNNs to learn surface statistical regularities arXiv:1711.11561.

Karpathy, A..

Kelly, D., Sanders, R., et al., 2008. Assessing the quality of scientific software. First International Workshop on Software Engineering for Computational Science andEngineering. Citeseer.

Kim, B., Wattenberg, M., Gilmer, J., Cai, C., Wexler, J., Viegas, F., Sayres, R., 2017. Interpretability beyond feature attribution: quantitative testing with concept activation vectors (tcav) arXiv:1711.11279.

Kim, J., Feldt, R., Yoo, S., 2018. Guiding deep learning system testing using surprise adequacy arXiv:1808.08444.

Krishnan, S., Franklin, M.J., Goldberg, K., Wu, E., 2017. Boostclean: automated error detection and repair for machine learning arXiv:1711.01299.

Krishnan, S., Wang, J., Franklin, M.J., Goldberg, K., Kraska, T., Milo, T., Wu, E., 2015. Sampleclean: fast and reliable analytics on dirty data.. IEEE Data Eng. Bull. 38 (3), 59–75.

Krishnan, S., Wang, J., Wu, E., Franklin, M.J., Goldberg, K., 2016. Activeclean: interactive data cleaning while learning convex loss models arXiv:1601.03797.

Kurakin, A., Goodfellow, I., Bengio, S., 2016. Adversarial examples in the physical world arXiv:1607.02533.

Lahoti, P., Weikum, G., Gummadi, K.P., 2018. Ifair: learning individually fair data representations for algorithmic decision making arXiv:1806.01059.

Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., et al., 2018. Deepgauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 120–131.

Ma, L., Zhang, F., Sun, J., Xue, M., Li, B., Juefei-Xu, F., Xie, C., Li, L., Liu, Y., Zhao, J., et al., 2018. Deepmutation: mutation testing of deep learning systems arXiv:1805.05206.

Ma, L., Zhang, F., Xue, M., Li, B., Liu, Y., Zhao, J., Wang, Y., 2018. Combinatorial testing for deep learning systems arXiv:1806.07723.

Ma, Y.-S., Offutt, J., Kwon, Y.R., 2005. Mujava: an automated class mutation system. Softw. Test. Verificat. Reliab. 15 (2), 97–133.

Masuda, S., Ono, K., Yasue, T., Hosokawa, N., 2018. A survey of software quality for machine learning applications. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 279–284.

McDaniel, P., Papernot, N., Celik, Z.B., 2016. Machine learning in adversarial settings. IEEE Secur. Privacy 14 (3), 68–72.

McKeeman, W.M., 1998. Differential testing for software. Digit. Tech. J. 10 (1), 100–107.

Moosavi-Dezfooli, S.-M., Fawzi, A., Frossard, P., 2016. Deepfool: a simple and accurate method to fool deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2574–2582.

Sculley, D., Phillips, T., Ebner, D., Chaudhary, V., Young, M., 2014. Machine learning: the high-interest credit card of technical debt, https://www.eecs.tufts.edu/~dsculley/papers/technical-debt.pdf

Murphy, C., Kaiser, G. E., Arias, M., 2007. An approach to software testing of machine learning applications, Proceedings of the 19th International Conference on Software Engineering & Knowledge Engineering, Technical Program.
Nidhra, S., Dondeti, J., 2012. Black box and white box testing techniques-a literature review. Int. J. Embed. Syst. Appl. (IJESA) 2 (2), 29–50.
Nie, C., Leung, H., 2011. A survey of combinatorial testing. ACM Comput. Surv. (CSUR) 43 (2), 11.
Odena, A., Goodfellow, I., 2018. Tensorfuzz: debugging neural networks with coverage-guided fuzzing arXiv:1807.10875.
Murphy, C., Kaiser, G. E., Hu, L., 2008. Properties of machine learning applications for use in metamorphic testing, Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering, Technical Program.
Olson, P., 2011., https://www.forbes.com/sites/parmyolson/2011/03/15/the-algorithm-that-beats-your-bank-manager/
Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A., 2016. The limitations of deep learning in adversarial settings. In: Security and Privacy (EuroS&P), 2016 IEEE European Symposium on. IEEE, pp. 372–387.
Pei, K., Cao, Y., Yang, J., Jana, S., 2017. Deepxplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles. ACM, pp. 1–18.
Qi, Z., Wang, H., Li, J., Gao, H., 2018. Impacts of dirty data: and experimental evaluation arXiv:1803.06071.
Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., Dennison, D., 2015. Hidden technical debt in machine learning systems. In: Advances in Neural Information Processing Systems, pp. 2503–2511.
Selsam, D., Liang, P., Dill, D.L., 2017. Developing bug-free machine learning systems with formal mathematics arXiv:1706.08605.
Sen, K., Marinov, D., Agha, G., 2005. Cute: a concolic unit testing engine for c. In: ACM SIGSOFT Software Engineering Notes, 30. ACM, pp. 263–272.
Strickland, E., 2016. Doc bot preps for the or. IEEE Spectr 53 (6), 32–60.
Sun, Y., Huang, X., Kroening, D., 2018. Testing deep neural networks arXiv:1803.04792.
Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D., 2018. Concolic testing for deep neural networks arXiv:1805.00089.
Tian, Y., Pei, K., Jana, S., Ray, B., 2018. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering. ACM, pp. 303–314.
Wang, J., Sun, J., Zhang, P., Wang, X., 2018. Detecting adversarial samples for deep neural networks through mutation testing arXiv:1805.05010.
Weyuker, E.J., 1982. On testing non-testable programs. Comput. J. 25 (4), 465–470.
Xie, X., Ho, J.W., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y., 2011. Testing and validating machine learning classifiers by metamorphic testing. J. Syst. Softw. 84 (4), 544–558.
Xie, X., Ho, J.W., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y., 2011. Testing and validating machine learning classifiers by metamorphic testing. Journal of Systems and Software 84 (4), 544–558.
Xie, X., Ma, L., Juefei-Xu, F., Chen, H., Xue, M., Li, B., Liu, Y., Zhao, J., Yin, J., See, S., 2018. Coverage-guided fuzzing for deep neural networks 3 arXiv:1809.01266.
Zafar, M.B., Valera, I., Gomez Rodriguez, M., Gummadi, K.P., 2017. Fairness beyond disparate treatment & disparate impact: learning classification without disparate mistreatment. In: Proceedings of the 26th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, pp. 1171–1180.
Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S., 2018. Deeproad: gan-based metamorphic autonomous driving system testing arXiv:1802.02295.
Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., Zhang, L., 2018. An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 129–140.
Zhou, B., Sun, Y., Bau, D., Torralba, A., 2018. Revisiting the importance of individual units in cnns via ablation arXiv:1806.02891.

Foutse Khomh is an associate professor at Polytechnique Montreal and FRQ-IVADO Research Chair on Software Quality Assurance for Machine Learning Applications. He received a Ph.D in Software Engineering from the University of Montreal in 2010, with the Award of Excellence. His research interests include software maintenance and evolution, cloud engineering, empirical software engineering, machine learning systems engineering, and software analytic. His work has received three ten-year Most Influential Paper (MIP) Awards, and four Best/Distinguished paper Awards. He has served on the program committees of several international conferences including ICSM(E), SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as JSS, EMSE, TSC, TSE and TOSEM. He is program chair for Satellite Events at SANER 2015, program co-chair of SCAM 2015, ICSME 2018, PROMISE 2019, and ICPC 2019, and general chair of ICPC 2018. He is on the steering committee of SANER, ICPC (chair), and ICSME. He initiated and co-organized the Software Engineering for Machine Learning Applications (SEMLA) symposium (https://semla.polymtl.ca/) and the RELENG (Release Engineering) workshop series (http://releng.polymtl.ca). He is an Associate Editor for IEEE Software.

Houssem Ben Braiek is a Ph. D student in the Computer Science and Software Engineering department at Polytechnique Montreal. He has industrial research experiences in different Canadian companies including Bombardier, Ubisoft and TAS. He received a Master Degree in Software Engineering from Polytechnique Montreal. He is nominated for the Best Master Thesis Award 2019 in Polytechnique Montreal. He received an Engineer Degree in Software Engineering with excellence award from National Institute of Applied Science and Technology, which is affiliated with the University of Carthage. His research interests lie in the area of Quality Assurance of Machine Learning-based Software Systems.