Testing ML-based Systems

Version 0.5

December 2021

Supported by:

Federal Ministry for Economic Affairs and Energy

on the basis of a decision by the German Bundestag

Authors: *Jürgen Grossmann* (Fraunhofer FOKUS), *Hans-Werner Wiesbrock* (IT-Power Solutions), *Mariele Motta* (Neurocat)

Content

1. Introduction

Machine Learning (ML) and especially the application of neural networks has been able to achieve amazing successes in recent years due to the availability of large amounts of data as well as the increase in computing capacity. These successes include applications from image recognition, which now achieve better results than humans in many areas, the almost human-like abilities of speech recognition and conversation, which were finally demonstrated convincingly by the NLP model GPT3, or the massive superiority of algorithmic decision systems in learning and playing strategic games such as Go, demonstrated by the Google subsidiary DeepMind.

The ability of a computer program to learn has been defined by Mitchel (Mitchell, 1997) in the following way:

*A computer program learns from an experience **E** with respect to a class of tasks **T** and a performance indicator **P** if its performance on tasks from **T**, measured by **P**, improves with the perception of **E**.*

Machine Learning is used as generic term for a sub-field of artificial intelligence, whereby the IT system is supposed to find solutions to problems on its own by learning from the information made available to it, in order to subsequently apply what it has learned to new data. Examples of ML algorithms are regression models, decision trees, Bayesian inference and kernel-based methods.

Typically, a differentiation is made between supervised learning, unsupervised learning and reinforcement learning. Typical areas of application for the latter are real-time decisions, navigation for robots, game playing, and all areas in which the independent acquisition of knowledge and skills is involved [19]. Supervised and unsupervised learning can in turn be divided into two sub-parts, each of which has its own characteristic applications. The two paradigms classification and regression can be assigned to supervised learning. Typical applications for classification are fraud identification, image recognition, customer behaviour analysis and diagnosis. Regression is more typically used for popularity prediction in advertising, weather forecasting, market prediction, lifetime estimation and population growth prediction. Unsupervised learning can again be divided into two sub-paradigms: dimensionality reduction and clustering. Typical applications for the former are big data visualization, compression, structural analysis, feature minimization. Characteristic of clustering are recommendation systems, targeted marketing, segmentation.

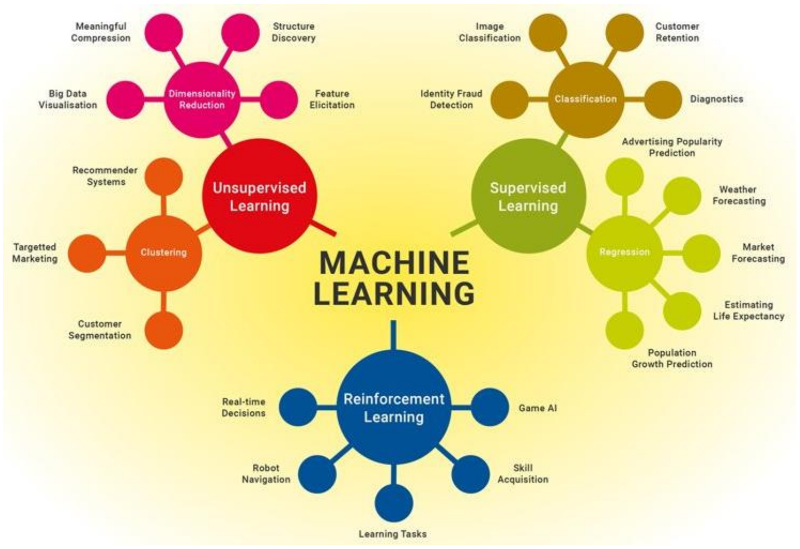

Figure 1 Different areas in ML and their fields of application

The success of ML in recent years has been particularly due to advances in the development of Artificial Neural Networks (ANN). ANNs define a ML approach that uses a layered network of mathematically modelled neurons. If an ANN has more than one internal layer (so called hidden layer), it is referred to as a Deep Neural Network (DNN).

Learning in the field of ML is conceptually related to the idea of optimization and the two terms are often used interchangeably.

With the increasing success of ML and ANNs, the need to integrate ML and ANNs into software systems that operate in a safety-critical environment is also growing. At this point at the latest, the question arises as to how ML and ANN-based systems can be tested and made safe. In this paper, we describe the basic challenges and requirements for testing ML-based systems, mainly in the area of supervised learning, and hope to make a first contribution to solving this problem.

2. General conditions of testing ML-based Systems

An ANN is a software system that differs from classical software in that the structural design of the software (i.e., the human-coded behaviour) does not determine the desired functionality. The structural design is quite simple compared to classical software and consists of a specific arrangement of parameters and algorithms in a graph structure. Parameters and algorithms are arranged in such a way that they are able to approximate the function desired by the user as accurately as possible within the framework of an optimization process based on data.

In particular, it is the data, the architecture of the network and the way how the training is carried out that are critical to the success of the approximation process. This dependence on data and architecture and the lack of function specific software code has both a major impact on quality assurance in general and testing in particular.

- The software code of an ML model is generic and can be considered quite simple. Thus, it usually does not show the same error probability that classical software has.
- On the other hand, the parameter settings that result from the training process and their interaction during inference are extremely complex and usually not

comprehensible to humans. They can be considered as a major origin of failures, but they are nearly impossible to test on a systematic basis.

As a result, a much broader scope has to be set for testing. In addition to the typical white and black box procedure, data and the training process must become the subject of more intensive testing.

In the following, we will take a closer look at some aspects that also define the general conditions for testing ML-based systems.

2.1. ML-Model and ML-based systems

In the context of quality assurance and testing, we cannot consider ML models in isolation. ML-models are trained, integrated, and applied within a particular technical and often physical environment. There is an extremely strong binding between the ML model and the environment in which it was trained and for which it was trained. Unlike classical software, this binding is often difficult to characterize because models and their properties are so complex that they are usually not understood.

- ML models are dependent on the input data and their pre-processing. The collection and pre-processing process is done by hard- and software components that thus has a major impact on the performance of the model.
- ML models provide complex output that must be carefully interpreted to lead to a reliable prediction or decision. This is usually done by additional software components that post-process the inference result.
- ML models might be safeguarded and monitored by dedicated software components to ensure a reliable performance over time.
- Finally, ML models are trained for a specific purpose, targeting a dedicated operational environment. Deviations between the environment (i.e. the data) used for training process and the actually operational environment might have crucial effects on the performance of the ML model in operation. Thus, especially the training process must be subject to quality assurance.

As a foundation for testing ML-based systems we distinguish the following terms.

An **ML-model** is the result of a training process that is dedicated to fulfilling a certain task or functionality. It processes a set of input data to support a decision-making process based on that input data and a previously learned state. ML-models are used for different tasks. In general terms these are regression, classification, clustering, dimensionality reduction and control tasks (Zhang et. al. 2019).

A **decision-making process** is a process based on assumptions of the target environment and a set of data that represent a state of the target environment resulting in the selection of a course of action among several possible alternative options.

A **training process** is the process of building an ML-model using a dedicated training infrastructure and a set of data. The training process consists of various activities to select and prepare the data required for training and to tune the parameters in a model structure with the aim of training an ML model that is able to generalize beyond the training data to previously unseen data.

The **training infrastructure** is a software-based infrastructure that enables an efficient training process. It consists of software that supports data selection, data preparation and the compilation of suitable data sets. It also provides algorithms and software to realize different model architectures and operationalizes the training process so that different candidate models can be generated and compared.

As already mentioned, data is central to the training process. In general, the following **data sets** are distinguished.

- **Training datasets** are datasets with examples used for learning the patterns and relationships in the data and are used to train the weights of the ML model.
- **Validation datasets** are used to tune the hyperparameters of a model. In particular, they are used to prevent overfitting of the model to the training data.
- The **test data sets** are used after training to test the generalizability of the ML model. They are selected independently of the training data but should have the same probability distribution as the training data set.

Validation and test data sets belong to the training process and thus to the modelling activities. This is to be distinguished in principle from the analytical activities of testing and quality assurance as defined in this paper. The latter must be separated from the constructive activities both technically and organizationally. On the one hand, the analytical activities are much more far-reaching than just testing the basic performance criteria such as over fitting and generalization, and on the other hand, they require organizational independence in order to be able to implement trustworthy results.

As already mentioned, we are interested in testing ML models in application. In this respect, we are looking at so-called ML-based systems.

A **ML-based system** is a software system, that integrates ML-models in order to support decision-making processes based on these models.

2.2. Open context and technology

ML-based systems are usually used for tasks that cannot be efficiently solved by classical programming. These include, in particular, problems that are too huge or too complex to be completely specified. This applies, for example, to applications that perform object detection in an uncontrolled environment such as road traffic or the surveillance of a railway line. In this case, the term open context problem is used to specify the Operational Design Domain of such a software. Open context problems are ∞-complex and cannot be specified correctly in all details (Podey et. al, 2019). Any modelling is subject to assumptions that lead to an incomplete or unreliable deduction of the *purpose*, *context* and *realization* of the ML-based system. In addition, state of the art specification processes lack adequate specification means to model this kind of uncertainty in a meaningful way. Finally model representations of the problem (including the ML-model and thus the ML-based system) are necessarily incomplete, since without exactly knowing the *purpose* and *context* of a problem there is no way to specify and find representative data for all possible corner cases and the *realization* will lack a complete coverage of the ODD.

Overall, such a situation leads to the fact that the verification (does the system work as planned/specified) of a system can no longer be fully guaranteed and, in particular, the question of validation (does the system do the right thing) must become increasingly important.

2.3. Stochastic solution approach and deep learning

ML is considered to be a stochastic solution that is often applied to problems, that are intrinsically non-stochastic problems. The recognition of objects, for example, is in principle a deterministic and not a stochastic problem. Stochasticity comes into play because, as already said above, the available knowledge about the purpose and the context of the solution is limited. A stochastic and data-based approach is considered to overcome some of the problems that are associated with the given knowledge gap.

It is assumed that the lack of explicit knowledge about the variety of objects to be recognized can be compensated for by the availability of a sufficient number of examples that implicitly allow this knowledge to be extracted from the examples in the course of a training process. However, this comes at cost. Since no one knows the original distribution of the problem space, examples can only be selected based on a "best guess" about the configuration of the problem space. Moreover, deviations and errors are intrinsic to a stochastic solution approach. Thus, It therefore always has to be assumed that a stochastic solution cannot be completely correct in the deterministic sense. There will always be a “natural” error rate that must be accepted. The aim of the optimization process is to reduce this error rate to an acceptable level.

Moreover, approximation methods are only partially reliable, and the generalization capability of any ML solution is limited and susceptible to distribution shifts.

The technical implementation as ANN or DNN also has some properties that have to be considered separately in the test process.

- In contrast to other forms of ML (e.g., linear or logistic regression, the k-nearest neighbour algorithm, Bayesian classifiers, SVM) specifically DNN lack transparency and stability. While interpretable models allow a human user to understand at least parts of the decision-making process, DNNs often show a better performance but in the same time the inference procedure lacks interpretability and statistical evaluability. This means that for a human observer, even if he or she has access to the internals of the model, it is not comprehensible on the basis of which parameters and properties in the ML model a particular decision is made.
- Furthermore, especially DNN lack reliable information on the quality of a decision. Although classification or regression models provide prediction probabilities at the end of the pipeline (e.g., by softmax output), these may unfortunately be often misinterpreted as model confidence. However, a model can be uncertain in its predictions even if its Softmax output is high (Yarin, 2016). The provision of reliable statements on the uncertainty of a model decision, on the other hand, would make it possible to also design safety-critical applications more reliably. If reliable information on the decision uncertainty is provided in addition to the results, results with high uncertainty could be handled separately by higher-level systems or the user.
- DNNs are not necessarily robust and are vulnerable to intentional and random perturbations. This has been shown in multiple examples through so called Adversarial Examples and the vulnerability of deep learning in the presence of noise.

- DNNs cannot be easily fixed or reoptimized at any point, i.e. models may have to be completely rebuilt if deviations occur.
- Last but not least, ML-Models are integrated to form ML-based systems that may consist of a complex interplay between ML-Models and classical software. Considering the tolerances, errors and uncertainties that underlie the processing of data in ML models, the combination of several ML models and their interconnection results in a degree of complexity that far exceeds the complexity of classical software.

2.4. Fault and failure model for testing ML-based systems

In classical software testing, a distinction is made between the terms *failure*, *fault* and *error*. While the term failure describes the perceived manifestation of a fault, the term fault describes the internal state of the program that has led to the failure and the term error describes the human cause that led to the fault. The ISTQB distinguishes the terms as follows:

- **fault (or defect):** a flaw in a component or system that can cause the component or system to fail to perform its required function, e.g., an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system
- **failure:** deviation of the component or system from its expected delivery, service, or result
- **error:** a human action that produces an incorrect result.

The existence of a failure shows that a system does not work as expected. However, not every fault in a software system shows up by a failure. Faults may have no effect because of the way the software is used, or their effect may be reduced by the shielding or corrective intervention of other software functions so that they do not become apparent. Moreover, failures are not only the result of software errors, but can also be caused by environmental conditions.

Zang et al. (Zhang et al. 19) extend the notion of defect to ML by defining that an ML bug (or ML defect) refers to any imperfection in a machine learning item that causes a discordance between the existing and the required conditions. Compared to the definition of a fault, which refers to flaws in components or systems, the definition of Zang et al. extends to so-called ML items, which, in addition to the components and systems, also allow other items from the ML process (eg. data) as carriers of a fault.

Humbatova et al. (Humbatova et al. 2019) created a taxonomy of ML-faults based on interviews with academics and practitioners in the area of ML. On a high-level view, the taxonomy provides classes that locate ML faults in the different artifacts developed in an ML process. They distinguish on high-level between faults in the ML-model (Model), the API (API), the input data (Tensors & Input) and the training artifacts (Training). Figure 2 shows the high-level categories used by the taxonomy from Humbatova et al.

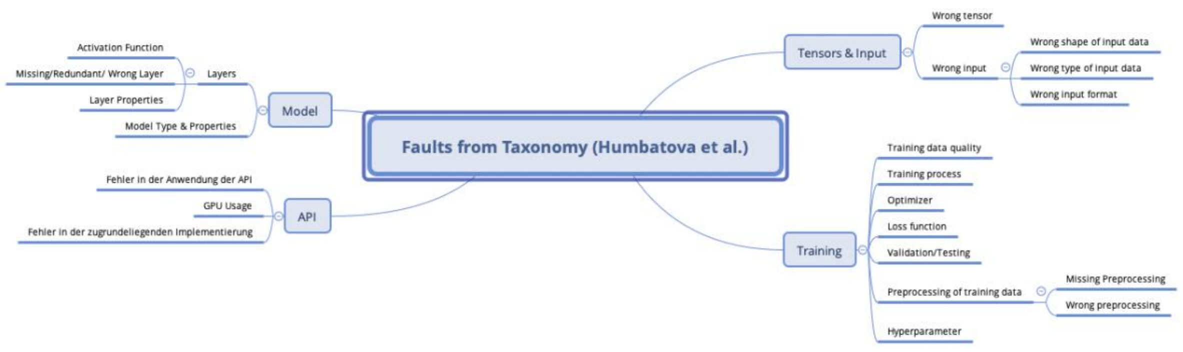

Figure 2 Categories of the ML fault taxonomy of Humbatova et al.

Even if the above terms and concepts can be applied to ML, it remains fundamentally necessary to extend them in such a way that the specifics of ML are addressed more strongly.

"Bug is not a suitable term to cover all functional insufficiencies, given its strong connotation to source code defects. Still, we need a new similarly succinct term in the context of MLware. We propose snag to refer to the difference between existing and required behaviours of MLware interwoven of data and source code. The root cause of a snag can be a bug either in the learning code or the infrastructure [36], but it is often related to inadequate training data – we call the latter phenomenon a dug." (Borg 2020)

Failures are usually identified as a deviation between the specification of a system and the actual behaviour of a system. As a prerequisite for such an approach, the specification of a system must be a reliable reference for the expected behaviour. Considering again the application of a system in an open context environment, the specification is not necessarily complete nor completely correct. In the automotive industry, for example, ISO 21448 (SOTIF) is concerned with ensuring the safety of intended functionality (SOTIF) in the absence of a failure. ISO 21448 applies to systems and applications that require adequate situational awareness to be considered safe and the term "absence of failures" is meant to characterize a system to act insufficiently even if it does not get into a specified failure situation. In addition to the absence of failures, such a system is expected to recognize potentially unknown and unsafe conditions and reduce the associated risks by itself. If it is not able to do so, the functionality or behavior is considered not sufficient for the aimed purpose.

Podey et al. (Podey et al. 2019) distinguish between the so-called

- **aimed purpose** of a system, which is implicitly expected and necessarily vague, and the
- **intended purpose** of a system, relating to explicitly expressed expectations that is for example given by a specification.

Podey et al. use the term **intended** in the same manner than **explicitly expressed** and as applied in the context of ISO26262 & SOTIF in the terms intended functionality and intended behaviour.

Furthermore, it must be asked whether a stochastic system, as we find it in machine learning, is not by definition subject to failures. ML is an optimization process that tries to approximate an aimed purpose by adapting a set of parameters to best fit with a given set of data. Separating the data in training, test and validation data sets helps detecting overfitting and allows to measure the generalization capabilities. The overall optimization process is a trade between different model characteristics and ensures that, on average, a model works as expected. However, this always implies that situations can be found in which a model decision does not represent an optimum or could even be considered as wrong. Whether these statistical deviations need to be considered as individual failures or not is currently not defined sufficiently.

How do we deal with a situation where an automated system on average leads to a better result than all comparable systems, but in a specific case performs in a clearly limited way? Do we consider this a failure and how do we deal with such a failure when we know that a better solution is not feasible under the given conditions?

Software reliability is for example measured by the probability of successful runs, which is defined as the sum of the probabilities of successful runs divided by the sum of the probabilities of all runs.

- **Operationalizability through statistical benchmarking. Question: Is the benchmarking correct?**
- **We cannot assign faults to every failure, but we can learn train.**
- **Do we need XAI for the identification of defect state and defect cause?**

2.5. Objectives of testing ML-based systems

Testing is the process of planning, preparation, and measurement with the aim of determining the properties of an IT system and showing the difference between the actual and the required state (Pol et. al. 2002).

Two things are remarkable about this definition. Firstly, the software specification is not emphasised as the sole reference for the desired behaviour of a system, as is the case in many other definitions. This means that the test does not only address deviations between the specification and the behaviour of a software system, but also possible errors in the specification. The goal of testing in this case is not only verification but also validation of the system. Furthermore, the definition restricts the actual activities of testing not only to dynamic test procedures that involve an executable test object but also to static quality assurance activities such as review and other forms of analysis. The latter seems to make sense especially for the testing of ML-based systems, since with data and data sets a quality-determining artefact exists that per se cannot be directly tested dynamically.

Primarily, software testing is an activity that tries to find faults. This can improve the overall quality of the system and reduce the likelihood of undetected failures occurring. Zang et al. have also formulated their definition of ML testing accordingly:

Machine Learning Testing (ML testing) refers to any activity designed to reveal machine learning bugs. (Zhang et al., 2019)

In modern software development, testing as a systematic component of quality assurance also fulfils other tasks. Testing, among other things, serves to build confidence in the functionality of a system. In addition to finding errors, this also includes systematic testing, which at least attempts to formulate arguments for the absence of errors under certain conditions. Systematic testing approaches are used for this purpose, which, for example, divide the input data of a software system into classes and test selected representatives of each class. Furthermore, testing can contribute to a better understanding of an unknown system. The idea of exploratory testing describes testing as a creative act in which the tester learns new properties of the SUT by performing tests and can incorporate them into new tests.

However, software testing cannot prove that the system is free from faults. According to Dijkstra "Program testing can be used to show the presence of bugs, but never show their absence!" The reason for that is that exhaustive testing is (except for very simple test objects) not possible. Exhaustive testing means that all software functions and also all possible values in the input data would have to be tested in all their combinations, which is practically impossible.

For this reason, various test strategies and concepts deal with the question of how to achieve a large test coverage with the smallest possible number of test cases.

Testing ML-based systems is the process of planning, preparation, and measurement with the aim of determining the properties of ML-based systems and showing the difference between the actual and the aimed state. (adapted from. Pol et. al. 2002).

In contrast to Zhang et al, we want to emphasize with this definition that testing ML is always also about testing the software surrounding the ML model. It is therefore not sufficient to ensure that an ML model works as intended as a single component, but always in the context of a practical application. This means that we test ML-based systems and not only ML models.

Riccio et al. state that “testing techniques should not solely expose misclassifications and prediction errors at the ML model level, but rather look at the side-effects of such inaccuracies at the overall system level. Individual misclassifications (or individual mis-predictions) are suboptimal definitions of failures if the whole MLS is considered, because they may have no consequences, or, on the contrary, may lead the overall system to deviate significantly from its requirements and result in a failure.”[5] (Riccio et al., 2019)

2.6. Verification vs validation

Challenges and limit of verification

Challenges in validation

3. Quality attributes addressed by testing ML-based systems

3.1. Model relevance

Overfitting, ability to generalize

3.2. Correctness

3.3. Robustness

Intentional perturbation, randomly expected perturbation

3.4. Efficiency

3.5. Security

3.6. Confidentiality, Integrity, Availability

3.7. Data Privacy

3.8. Fairness

3.9. Interpretability

3.10. Learnability

4. Test objects and integration in testing ML-based systems

The term test object describes the object to be tested by a particular analytical procedure. In the case of dynamic testing, this is also referred to as System Under Test (SUT), which somehow highlights the dynamic nature of the test object. However, in the following, we use the more general term test object, analogous to the ISTQB, analogous to the ISTQB, to emphasize that we are addressing both static and dynamic test procedures.

Although our primary test object, as the name of this paper suggests, is an ML-based system in a meaningful deployment environment, we obtain several other test objects that can be tested individually or partially integrated considering the genesis of an ML-based System as well as its systematic integration from individual components.

Due to the high importance of the data and the training process, we explicitly distinguish between test objects of the training phase, which are decisive for the quality and properties of an ML model, and the development and runtime artefacts, which are relevant for the software-technical creation and integration of an ML-based system based on individual components.

Zhang et al., 2019 for example distinguishes on a high-level between data, the learning program (i.e., the training infrastructure) and the ML-framework (i.e., the libraries and building blocks that are used to define models).

"Thus, when conducting ML testing, developers may need to try to find bugs in every component including the data, the learning program, and the framework." (Zhang et al., 2019).

From our point of view, the test objects of the training phase include at least the following artefacts.

- Data
- Hyperparameters
- Loss function
- Optimiser
- Training KPIs and end criteria
- Network architecture

The test objects of the software-technical creation and integration are:

- Data pre-processing software during engineering
- ML-Model
    - Parameters
    - Implementation of the model by software
- ML-Framework
    - Libraries and algorithms
- Data pre-processing for operation
- Model in integration
    - Safety mechanisms (safety cage, redundant models)
    - Software-hardware embedding (model and data pre-processing or result preparation, GPU integration)
- ML-based system or component (full integration)

However, the concrete definition of individual test object is determined by the test method used, the life cycle phase in which the test takes place and by the progress of the integration of an ML-based system from its individual components.

Especially in dynamic testing, the definition of the test object depends strongly on the test levels in the integration. Classically, a distinction is made here between the component test, the integration test, the system test and the acceptance test. Transferred to the test of ML-based systems, we have to consider the following test levels.

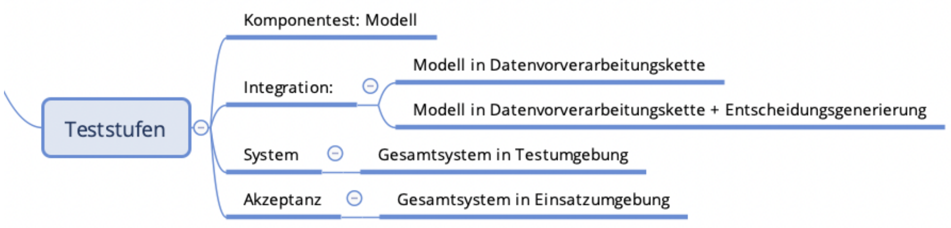
Teststufen
Komponentest: Modell
Integration:
Modell in Datenvorverarbeitungskette
Modell in Datenvorverarbeitungskette + Entscheidungsgenerierung
System
Gesamtsystem in Testumgebung
Akzeptanz
Gesamtsystem in Einsatzumgebung

Figure 3 Different test levels in testing ML-based systems

- During data testing the data that are used for training are tested.
- During component testing, the individual components of an ML-based system are tested. These include the ML model, the data pre-processing, all other functional components as well as components that are explicitly introduced to secure the runtime of an ML model.
- During integration testing, interface functionality and integrated behaviour are tested. During integration, the ML model must always be tested in the data processing chain. This means that the ML model is integrated with any components for data pre-processing as well as in the integration with post-processing or decision-making. In addition, it must be checked that the ML model and the ML framework are tested regarding all other aspects of hardware and software integration.
- During system testing the whole ML-based system is tested in a dedicated test environment.
- During acceptance testing the whole ML-based system is tested in its actual operation environment

Furthermore, it would have to be discussed whether, due to the risks associated with the uncertainties of ML, online tests should not also be taken into account at runtime. These tests go hand in hand with dedicated security and monitoring components that are supposed to identify corner cases and potential distribution shifts.

5. Test methods for testing ML-based systems

This chapter describes various test methods that can be used for analytical quality assurance of ML-based systems. A recurring question is whether testing should be performed exclusively dynamically, i.e. the executable, compiled code is always checked, or whether static analyses such as reviews, model checking, etc. should also be counted as testing.

In the case of ML-based systems, where the meaning of individual parameters, for example, is no longer clear and which can only be adapted by extensive learning on the basis of training data, static methods are, in our view, an essential part of testing. Completeness and representativeness of the data, for example, cannot be evaluated simulatively, and neither can the relevance of a model.

Since ML-based systems are in particular also SW systems, the known and established test methods can also be applied to them. Due to their special characteristics, however, some of these methods are more or less relevant and may need an adaptation here and there. In the following, some common methods are briefly described, which modifications may have to be made, and evaluated for their suitability for testing these new systems.

5.1. Requirements-based testing

The standard norm for testing is the fulfilled proof of all requirements in a requirement specification. ML-based systems, however, usually operate in open domains, see chapter 2.2. An even approximately complete list of requirements will not exist. Therefore, explicit requirements are usually insufficient as a basis for testing. Nevertheless, a detailed analysis of the requirements specification should be the starting point for testing. Because in it the delimitation of the ML-based system is to be found, what it should do and where it is not responsible, and above all, in which environment it is to be used. Very helpful, if available, are detailed use cases [9], which narratively describe requirements and the possible interactions of the system with their intended environment.

5.1.1. Evaluation of the data

To evaluate the training data, its balance and completeness, it is essential to obtain a good overview of the various contexts in which the ML-based system is to be used. Therefore, it recommended to first extensively analyse and extend the given use cases. These virtual simulations can more precisely delineate the operation domain and identify further edge cases. Thus, they help to develop an ontology of the environment.

5.1.2. Derivation of test cases

Based on the requirements in the specifications, test cases must be systematically derived that can provide evidence of their fulfillment. The guiding question here is what can go wrong and will the identified, possible errors be completely covered up by tests.

Since ML-based systems usually infer probabilistically, an error cannot be proven by one counterexample, e.g. one wrong face recognition. Rather, it has to be checked during testing whether statistically the number of false results is significant. Only then is there a case of misbehavior. This means, in particular, that in the case of ML-based systems, a single proof of an erroneous inference is not sufficient. If necessary, many similar tests must be defined and performed for a requirement so that a statistical evaluation of their results is possible.

5.2. Risk-based testing

Increasingly with their successes and growing capabilities, ML-based systems are also being used in critical areas. A prerequisite for this is a risk analysis. In the risk analysis, extensive testing is then required, especially for risk mitigation, to prove possible, dangerous behavior of the system as unlikely.

In some of the general risk drivers and assesments such as Security and Safety or item types ML-based systems are not fundamentally different to general SW, [11], and shall therefore not be considered further here. In the area of functional safety and especially in the possible risk exposure and analytical estimation, however, they are very different.

5.2.1. Functional Safety

Again, due to the statistical nature of the ML-based system, a single counterexample may not immediately be considered a violation of Functional Safety. Rather, the failure must be statistically proven here as well.

Various factors influence the estimation of ML technology-related risks: risk exposure in the environment, severity of the hazard and statistical behavior of the ML-based component. In order to now estimate the remaining risk using testing, it is recommended to represent the results by means of a 3-shaped risk tensor, where the indices go over the above factors, [10]. A final overall assessment of the risk can be performed with this tensor application specific.

5.2.2. Analytical Estimation

Dynamic tests always check samples whose selection must also be justified. Therefore, in the field of conventional SW, complementary analytical estimations are usually expected. However, this is almost impossible for ML-based systems that use e.g., deep nets, since they are hardly analytically accessible so far with their multitude of parameters and nonlinearities. Current Explainable AI (XAI) research is investigating many different approaches here.

Over- Underfitting.

If the ML model has too many or too few parameters, new misbehaviour may occur. In the first case, the system learns to react convincingly in the training situations, but fails in new situations, (Poor Generalization, Overfitting). In the second case, the system only moderately learns the training data, but usually behaves similarly in new situations (Underfitting). Thus, the system will react much more imperformantly in novel situations or will consistently fail to exploit a possible mitigation potential. Indicator for overfitting or underfitting is monitoring and evaluating the learning process. If, in the further course of training, the loss values on the validation set are significantly higher than on the training data, this indicates overfitting. If, on the other hand, the loss values on both data sets remain equally bad despite extensive learning, this points to underfitting.

Analytically, behind the problem of over/under fitting lies the fact that the capacity of the network is not adequate to the problem (relevance of the model). For this purpose, the capacity of the selected ML model can be estimated analytically [12] and compared with the complexity of the learning data.

Lipschitz constraint

If a natural metric exists on the space of learning data, e.g., a variant of the Euclidean metric on the high-dimensional real vector space of pixel data of images, and the decision of the ML model is based on simple evaluations in a high-dimensional feature space, e.g., via max-pooling or softmax in the penultimate layer of a neural network, a Lipschitz constraint can be usefully defined and estimated. The Lipschitz constant then provides an analytical estimate of a network's susceptibility to perturbation and hence its robustness.

5.3. Search-based testing

Are runtime constraints met, memory consumption, or other constraints such as value or output ranges? These are classic questions to the test, which are met by a targeted search for extreme examples. Various optimization algorithms are used to solve them. For ML-based systems, the use of these methods shifts away from the possible extreme cases for the test object to targeted perturbations of the inputs for the purpose of intentional misbehaviour. E.g., how to purposefully obtain false classifications, manipulate facial images so that the account protected with biometric data can be opened without authorization? Or in the case of reinforcement learning, how are the feedbacks to be changed so that the algorithm becomes unstable?

Since ML-based systems are now used in many security-critical areas, robustness against hostile manipulation is of utmost importance. Thus, there is a whole branch of research that investigates the topic of adversarial attacks [15].

In the classic case of search-based testing, individual extreme examples are sought for the test. The goal is to test the test object even in these cases. This approach reaches a new level for ML-based systems. Since the learning process of these systems is typically an optimization itself, this technique can be used not only to discover single counterexamples, but also to generate arbitrarily many new input data that can challenge the old system. For this purpose, one couples a generative model and a discriminator together in the form of a two-person zero-sum game (GAN): the generator tries to produce as good as possible fake copies of the original data, the discriminator hangs to distinguish real and fake data [14], which are available with the training data.

Nowadays, there are various extensions of this technology and a wide variety of applications, including transforming a blackbox ML-based system into a whitebox system with a known architecture.

5.4. Combinatorial testing

A common way to systematically increase the test depth is to combine different test cases. In the area of embedded systems, for example, the classification tree method [13] is often used for this purpose. The initially unstructured test space is partitioned into classifications and classes according to various test aspects. This partitioning is then used for test planning: Which combinations of classes are relevant, which combinatorics should be considered, and how many test cases should be defined for them?

This technique gains new importance for testing ML-based systems.

Since the meaning of the numerous parameters of a model and their influence are usually not understood and are only adjusted by the learning process, a balanced, representative selection of training data is essential. In order to check the completeness of the data it is useful to define the planned environment of the test object according to various aspects like: Typical situations, risk areas, possible disturbing influences…. at first. Afterwards this classification can be refined into concrete scenarios and risks or also perturbations e.g. and in this way the environment can be divided according to various viewpoints. Such a partitioning of the environment, weighted with their respective meanings, the frequency of their occurrence, the associated risk, etc., then allows a combinatorial estimation of the required distribution of the training data.

5.5. Metamorphic testing

5.6. Exploratory testing

5.7. Probabilistic testing

5.8. Testing with failure models

5.9. Diversifying test

Back to back test, regression test

5.10. Reviews

5.11. Static analysis

6. Workflow and process aspects of testing ML-based systems

Software testing is an activity that is oriented towards the life cycle of a software component and itself passes through various phases. A typical software testing process can for example distinguish *test planning*, *test design & analysis*, *test implementation & execution* as well as *evaluating test exit criteria and reporting*.

These or slightly varying phase distributions are also found in the literature and common standards. In the context of this paper, we use the above-mentioned division into phases to describe the challenges along the individual phases that we see specifically for the testing of ML-based systems.

6.1. Test planning phase

Roughly speaking, the test planning phase serves to define the quality objectives, determine the test objects and set up a test strategy that serves to test the desired quality objectives in a meaningful way. Afterwards the entire test process is planned in its technical, temporal, and monetary aspects, taking into account the available resources.

A test strategy describes which parts of the system are to be tested with which intensity, using which test methods and techniques, using which test infrastructure and in which order.

Testing ML-based systems places some special challenges on the test planning phase.

Challenge 1: Selection of appropriate quality and test objectives

Since ML-based system slightly differ in terms of engineering as well as operation, the test process must address additional test objectives, that are often not addressed in classical software testing. Besides coverage of the relevant functional aspects of the application context including standard cases/scenarios, all critical corner cases/scenarios as well as all defined non-functional properties like security, robustness, performance etc., testing ML-based systems need to reveal

- data and labelling errors that lead to critical functional failures
- software failures that undermine critical functionality during model training and model inference
- unused or unintended decision capabilities of a model
- bias and noise in decision processes
- known vulnerabilities and failure modes of the technology used eg. in DNNs/CNNs

Challenge 2: Determining all relevant test objects and the corresponding test procedures

To comprehensively test ML-based software systems, several new test objects must be considered that are given little to no attention in classic software. These test objects are:

- data and labels
- hyperparameters
- loss function
- optimiser
- training KPIs and end criteria
- network architecture and additional design decision defining basic model properties
- the ML-Model including the software implementation of the models' internal behaviour and all parameter settings
- the ML-Framework including the used libraries and algorithms

- data pre-processing software during engineering
- additional components that serve a proper integration of the ML-model including safety mechanisms (safety cage, redundant models), model and data pre-processing or result preparation, GPU integration.

Challenge 3 Definition of an appropriate integration and test procedure.

ML-based systems are complex entities with high dependencies. Thus, the quality of an ML-based decision system is not only based on the performance of the ML model, but also on

- the performance of the data pre-processing chain including all the required sensors and data fusion components,
- the software that interprets the output of the ML model, processes it for humans and/or translates it into actions, and
- the seamless interaction of all these components.

In addition, the quality of the target system is dependent on the training data, data preparation, and training infrastructure. Thus, a systematic test approach does not only target the system and its integration, but also the entire data acquisition and training infrastructure. If we take this into account, the test levels of classical software testing can be extended as follows.

- data pipeline testing
- training pipeline testing
- data and data integration testing:
- component testing: ML-Model, data pre-processing, decision making
- integration testing: Model in data pre-processing chain, Model in data pre-processing chain + decision making, ML-model subsystem with safeguarding
- system testing: Entire system in test environment
- acceptance testing: Entire system in operational environment
- runtime testing

6.2. Test design & analysis phase

The test design and analysis phase serve to implement the test objectives defined in the strategy in a meaningful way. This includes the identification of the abstract tests, the definition of suitable coverage and completeness measures and the specification of suitable procedures and frameworks for the automation of the tests.

Challenge 1: Identification of appropriate data testing procedures

Due to the high importance of data for the performance of a ML model, both the data, its origin, its storage, and preparation must be systematically tested and reviewed. In this context we distinguish between testing the data acquisition, preparation and storage infrastructures and testing the data and data quality itself.

Testing the data acquisition, preparation and storage infrastructures mainly addresses aspects of infrastructure testing like data base testing, testing the underlying communication and computation platforms regarding performance and availability, and the data processing infrastructures that allow for data preparation and refinement. The test approach must consider that these infrastructures are often dealing with big data that is, most of the processes are highly automated and require a high degree of availability and scalability that poses special requirements on hardware and software solutions with corresponding challenges for testing (see [16][17]).

According to L.P. English [18]data quality can be subdivided into three aspects, which can be considered independently of each other.

- Data definition and information architecture quality describes the quality of the data specification based on the application context.
- Data content quality describes the inherent quality characteristics of the data such as correctness of data values, completeness, unambiguity, freedom from errors, etc.
- Data presentation quality describes how the data can be made available appropriately quickly, in a suitable format, and with a reasonable amount of effort.

Data quality dimensions are attributes of data quality that, if measured correctly, can describe the overall level of data quality. The identification of relevant quality dimensions forms the basis for the assessment and subsequent improvement of data quality. The quality dimensions are usually highly context-dependent, and their relevance and importance can vary depending on the organization and data type. The most common, i.e., the most frequently cited dimensions in the literature, are completeness, timeliness, and accuracy, followed by consistency and accessibility [19].

Overall, assessing data quality for ML applications is a complex task. Current best practices suggest that more data and better models provide better results.

- Poor data quality can cause significant problems in both ML model building and big data applications.
- Certain systematic preprocessing operations on the data help these models achieve higher effectiveness.
- While traditionally data quality is assessed before the data is used, in the machine learning context quality can be assessed both before and after the model is built.
- Data quality can be assessed before the learning process along the data and its compilation processes and after the learning process along the performance of the ML model.
- The data quality is evaluated along different quality attributes, so that systematic evaluation criteria for the data quality can be established.

To date there are no testing approaches that directly address the issues from above in a systematic and automated manner.

Challenge 2: Identification and selection of appropriate tests for complex/open world scenarios

Testing machine learning suffers from a particularly difficult form of the oracle problem. While classical systems are usually fully specified, machine learning systems are designed to provide meaningful answers to questions for which there is not yet an answer known [1] (Zhang et al.). Training ML models typically aims to achieve good performance on training data while being able to generalize well to unseen, new data. For the models to learn the underlying function from the data provided to them, that data must sufficiently capture the features of the real-world problem. If incomplete, outdated, or irrelevant data are provided to the model, the model will not generalize towards unseen data.

The problem for testing then consists of defining suitable criteria for defining the completeness of the data for a partially unknown range and to generate test cases that systematically represent the entire input range. In addition, the test cases must be stored with suitable expected values that allow a systematic evaluation of a test run. This special form of the Oracle problem known from testing prevents a scalable test data generation.

Solution approaches, such as metamorphic testing [32], are not yet able to realize the necessary scalability and efficiency required for a comprehensive testing approach.

Challenge 3: Dealing with ML-specific failure modes

Since ML and ML-based systems show significant differences to classical software engineering, testing processes may fail if they do not address failure modes that are specific for ML-based systems. These failure modes include bias, non-determinism, lack of robustness, and lack of transparency and understandability.

Decision bias: Bias in machine learning is a type of error in which certain elements of a dataset are weighted and/or represented more heavily than others. A biased dataset does not accurately represent the intended use case of a model, leading to biased results, low accuracy, and analytical errors. Bias can occur in several different areas, from human reporting and selection bias to algorithmic and interpretation bias. Sampling bias, for example, occurs when a dataset selected for training does not reflect the realities of the use case (e.g., when facial recognition relies significantly on data from only one population group e.g., men, women, Europeans). Exclusion bias most often occurs in the pre-processing phase of the data. It is often caused by the deletion of valuable information that is considered unimportant e.g., the deletion of a relevant feature that has not been recognized or that has been considered as unimportant. Measurement bias occurs when the data collected for training is different from the data collected in the real world, for example, when different sensors are used to record the training data as with the production data. Measurement bias can also result from inconsistent label assignment during the data labelling phase of a project. Finally, observer bias also known as confirmation bias, is the effect of seeing what you expect or want to see in the data during manual data selection and labelling processes.

Probabilistic nature and non-determinism: ML-based software, even if it has some fundamentally deterministic properties, is not necessarily stable with respect to the environment and environmental changes. Moreover, the training process itself is often nondeterministic and thus difficult to reproduce. Non-determinism in the training phase arises from the random initialization of model parameters, the stochastic selection of training data (e.g. mini batch sampling), and the use of stochastic functions in the optimization process. Non-determinism in the operation phase may arise using stochastic activation and weight functions. Moreover, neural networks are typically trained on graphics processing units (GPUs), which, under certain experimental conditions, yield nondeterministic outcomes for floating point operations.

Missing robustness: Robustness is the ability of a computer system to deal with erroneous input and to handle errors during execution. An ML model is considered robust if small perturbations in the input space yield only small perturbations in the output space. Since ML has been shown to be especially vulnerable against so called adversarial examples and against distributional shift, it can only be considered robust under certain circumstances.

An adversarial example is an input to a neural network that has been modified in such a way that it alters the output of the neural network, even though a human would still recognize the original class. In the extreme case, the modified input is indistinguishable from the original input for a human. Distributional shift describes a difference between the test and training environments [Ref 1]. Such distributional differences can be considered as gaps in the representation of reality and are a general problem in designing ML applications to be used in real-world applications. If the perceptual or heuristic inference processes of such a model have not been adequately trained to the correct distribution or the distribution of the environment changes in operation, the risk of unintended and harmful behaviour increases significantly.

Lack of transparency and understandability: Neural networks function as black box systems. Instead of humans explicitly coding the system behaviour with conventional programming, in ML the computer program learns based on many examples that represent the mapping of the input data to the desired output. Transparency in AI is generally referred to as explainability, which includes both interpretability and confidence in the system and its genesis[29][30]. While interpretability is the degree to which a human can understand the cause of a decision [31], confidence in a system is gained by understanding the system itself, its operational environment as well as the development of the system.

A challenge regarding testing arises from the dependence on a system that not even the developers and testers really understand. To gain confidence and certainty regarding elemental quality properties of neural networks, it is essential to enable at least a certain degree of human interpretability and understandability.

Challenge 4: Definition of appropriate coverage and completeness criteria

Due to the lack of logical structures and system specification, it is still unclear how evidence regarding test completeness could be provided for ML-based systems especially for those with DNN components. To date, there are several proposals that combine systematic testing of ML-based systems with coverage criteria related to the structure of DNNs. These include simple neuron coverage by Pei et al. [23], which considers the activation of individual neurons in a network as a variant of statement coverage. Ma et al. [22] define additional coverage criteria that follow a similar logic to neuron coverage and focus on the relative strength of the activation of a neuron in its neighbourhood. Motivated by the MC/DC tests for traditional software, Sun et al. [24] proposes an MC/DC variant for DNNs, which establishes a causal relationship between neurons clustering i.e., the features in DNNs. The core idea is to ensure that not only the presence of a feature, but also the combination of complex features from simple feature needs to be tested. Wicker et al. [25] and Cheng et al. [26]refer to partitions of the input space as coverage items, so that coverage measures are defined considering essential properties of the input data distribution. While Wicker et al. discretizes the input data space into a set of hyper-rectangles, in Cheng al. it is assumed that the input data space can be partitioned along a set of weighted criteria to describe the operating conditions. Finally, Kim et al. [21] evaluate the relative novelty of the test data with respect to the training dataset by measuring the difference in activation patterns in the DNN between each input. A good summary of the current state of the art regarding coverage criteria for testing DNNs can be found in [20]. In addition, the work of Dong et al [27] claims that there is only a limited correlation between the degree of different kinds of neuron coverage and the robustness of a DNN, i.e., improving the degree of simple neuron coverage measures does not significantly contribute to improving the robustness. However, in their study, Dong et al. did not analyse the effect of more complex coverage approaches (e.g., feature coverage and the MC/DC variant for DNNs) as well as coverage approaches that address the partitioning of the input data space.

6.3. Test Implementation & execution phase

During the implementation and execution phase test cases are created and executed. Test cases should be based on the objectives and requirements identified during the planning and analysis phase. During the execution, the test team performs all tests. The deviations are logged, and defects are identified. Deviations are measured as the difference between actual and expected test results.

Challenge 1: Synthetic test data generation

ML systems process a wide variety of data. These range from simple tabular data to complex data streams (images, movies, radar or lidar data), such as those processed in ML-based perception systems. To be able to test such systems and to make the necessary large amounts of data available in sufficient diversity, data will have to be synthetically generated. The more complex the input data, the more complex is the process of data generation. For example, the creation of synthetic film sequences is significantly more complex and resource-intensive than the provision of simple numerical quantities.

Challenge 2: Achieving the necessary degree of automation and scalability.

The complexity and uninterpretability of DNNs lead to the fact that manual testing approaches are not sufficient to perform a comprehensive quality assurance of a DNN.

To cope with the complexity of the applications and to achieve consistent results in repeated tests a high degree of automation is required. Automation should encompass all necessary activities of the testing process, starting with test case identification, test data generation, test execution, and final test evaluation. Similar, to the training of an ML model, such an automated testing approach relies on a larger technical infrastructure that realizes automation in a in a trustworthy and reliable manner.

However, generating test cases automatically is still a challenge. For instance, studies [85, 86] claimed that the test cases generated by an automated testing tool may not cover all real-world cases. (Zhang 2020)

6.4. Evaluating exit criteria and reporting phase

The test evaluation and reporting phase is used to evaluate the test execution against the defined and agreed exit criteria. Based on this evaluation, a decision can be made as to whether enough tests have been performed to achieve the quality objectives defined in the planning phase. The result of the test evaluation is then documented and summarized in a form that can be understood by all relevant stakeholders.

Challenge 1: Define and apply appropriate end-of-test criteria and validation metrics.

The interpretation, aggregation and evaluation of individual test results and the evaluation of the entire test process for ML-based systems can differ greatly from the procedures that are established for classical software systems. On the one hand, completely new test procedures have to be taken into account due to the consideration of data as a decisive quality factor, and on the other hand, the specific characteristics of an ML-based system, especially with regard to its failure characteristics, lead to different evaluation approaches.

On the one hand, DNNs in particular feature a complexity that is not reached by classic software. While it is possible to trace failure modes back to individual errors in classical software systems, this is much more difficult in ML-based systems. The high number of parameters, hyperparameters and optimization decisions makes it almost impossible to identify wrong parameters as the cause of a concrete failure mode.

Additionally, when considering different quality properties, it is important to keep in mind that there are dependencies between these properties, so that improving the KPIs for one property will worsen the KPIs of another property.

Risk-based testing approaches are basically able to relate variable quality properties of a system to the risks to the financial and fundamental risks of an application. An end-to-end approach on how to comprehensively apply risk-based testing in the context of ML systems has been sparsely explored.

Challenge 2: Communicate test status and evidence on quality in a comprehensible and trustworthy way

Test reports are designed to enable managers and users of software products to assess and understand the quality and risks of a software product in its application. To this end, the tests, their results, and the metrics used to demonstrate the performance of an ML-based system must be expressed in terms of their impact on the application domain in an understandable way. This is particularly important when it comes to assessing interconnected quality properties between which there may be a conflicting objective.

7. Testing in the ML-Life Cycle

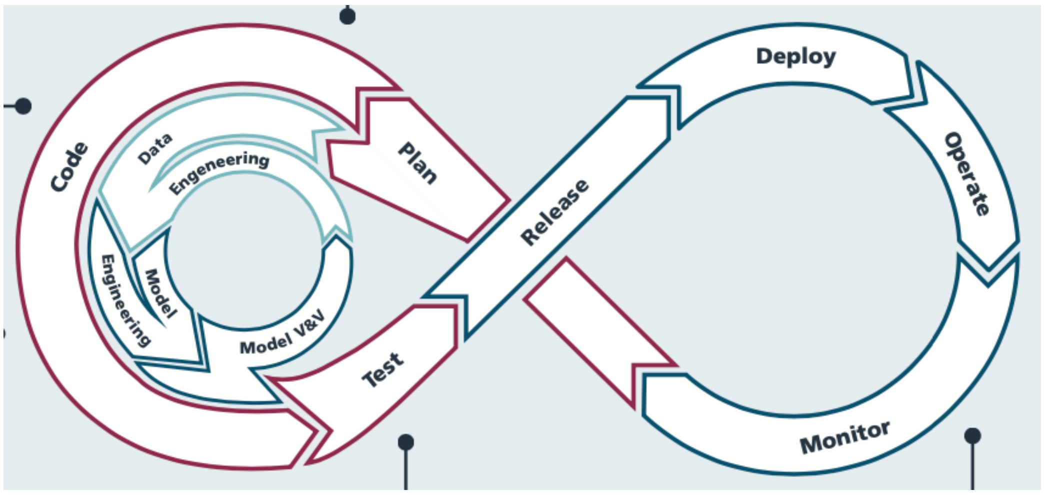

Figure 4 The MLOps life cycle

7.1. Plan

Purpose: *This phase serves to develop a basic understanding of the problem to be solved by ML as well as the data required for it. Developers and spzeificators decide which features are feasible with machine learning and which can be useful for a given existing or a new product. Importantly, it is at this stage that they decide what types of models are best suited for the given problem and what data is needed to successfully learn the desired features.*

- Testing requirements and KPIs

7.2. Data Engineering

This phase aims to identify the right data sets in the right distributions, collect the data in such a way that the model output can be delivered as efficiently as possible, enrich the data through labeling, store the lineage of the data, verify the quality of the labeled and prepared data, establish specific metrics to measure the quality of the data, store and analyze the data.

1. Testing the data sources and their reliability
2. Testing the data preparation and training pipeline (training phase)
3. Testing the data

7.3. Code

Purpose: *This phase aims to develop the source code that is required for setting up the model and related components. This includes, among other things, all components that form the data preparation and decision pipeline during deployment. The individual components, the software implementation of the model and the integration of all components into a functional unit must be realized and tested.*

1. Testing the decision pipeline (in integration)

7.4. Model Engineering

Purpose: *This phase aims to parameterize and train model variants based on the available data and their labels by considering all related requirements, and to identify and select the most appropriate model. In practice, training runs are performed with different model architectures and initial parameters (hyperparameters) and the results are compared. All resulting models are benchmarked, evaluating their properties in terms of accuracy and* *generalizability. The best model(s) is/are selected and passed to the subsequent process for further V&V. For modeling, predefined ML frameworks are usually used to support the developers in creating the model code and realizing the algorithms.*

1. Testing the model architecture and other design decisions
2. Testing the model pipeline
3. Testing the training assumptions (Hyperparameter settings, algorithms etc.)
4. Testing the model software and third party libraries

7.5. Model V&V

Purpose: *In this phase, the model is fully validated and verified. The aim is to assess the extent to which the model fully meets the functional and extra functional requirements given. In addition to accuracy and generalizability, scalability, complexity, robustness, fairness and resource requirements in particular play a crucial role in the evaluation of a model. The model is checked against all relevant performance characteristics and KPIs. Finally, it is decided at this point whether the model(s) at hand are suitable for integration into the software stack or need to be improved via further iterations.*

1. Testing the functional aspects of the ML-model
2. Testing extra-functional aspects of the ML-model e.g. robustness, fairness, transparency, security, scalability

7.6. Test

Purpose: *The selected model must be tested and evaluated in its deployment environment. For this purpose, the model must be integrated with the software and hardware of the target platform and systematically tested. The evaluation focuses on the safe functioning of the model in interaction with the necessary software and hardware as well as the interaction with additional safety components and functions, such as watchdogs for identifying dangerous situations and performing plausibility checks, redundancy for safeguarding the overall functionality, etc.*

1. Model integration testing
2. SW/HW integration testing

7.7. Release

Purpose:

1. Testing end to end functionality

7.8. Deploy

Purpose:

1. Acceptance testing

7.9. Operate

Purpose:

1. Runtime testing

7.10. Monitor

Purpose: *Collecting data and providing analytics on customer behaviour, performance, errors and more.*

8. Summary and conclusion

9. References

[1] Zhang, J. M., Harman, M., Ma, L. & Liu, Y. Machine Learning Testing: Survey, Landscapes and Horizons. arXiv:1906.10742 [cs, stat] (2019).

[2] Humbatova, N. et al. Taxonomy of real faults in deep learning systems. in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering 1110–1121 (ACM, 2020). doi:10.1145/3377811.3380395.

[3] Poddey, A., Brade, T., Stellet, J. E. & Branz, W. On the validation of complex systems operating in open contexts. arXiv:1902.10517 [cs] (2019).

[4] M. Pol, T. Koomen, und A. Spillner, Management und Optimierung des Testprozesses: ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap, 2., Aktualisierte Aufl. Heidelberg: dpunkt-Verl, 2002.

[5] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, und P. Tonella, „Testing machine learning based systems: a systematic mapping“, Empir Software Eng, Bd. 25, Nr. 6, S. 5193–5254, Nov. 2020, doi: 10.1007/s10664-020-09881-0.

[6] L. Myllyaho, M. Raatikainen, T. Männistö, T. Mikkonen, und J. K. Nurminen, „Systematic literature review of validation methods for AI systems“, Journal of Systems and Software, Bd. 181, S. 111050, Nov. 2021, doi: 10.1016/j.jss.2021.111050.

[7] Gal, Yarin. Uncertainty in Deep Learning, University of Camebridge, October 13th, 2016

[8] M. Borg, „The AIQ Meta-Testbed: Pragmatically Bridging Academic AI Testing and Industrial Q Needs“, arXiv:2009.05260 [cs], Sep. 2020, Zugegriffen: Okt. 13, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/2009.05260

[9] Jörn Müller-Quade et al., Sichere KI-Systeme für die Medizin, https://www.plattform-lernende-systeme.de/files/Downloads/Publikationen/AG3_6_Whitepaper_07042020.pdf

[10] Paul Schwerdtner et.al., Risk Assessment for Machine Learning Models, arXiv:2011.04328v1

[11] Michael Felderer et. al., A taxonomy of risk-based testing, arXiv:1912.11519v1

[12] Vladimir Vapnik et.al, Measuring the vc-dimension of a learning machine, Neural computation 1994.

[13] Matthias Grochtmann, et. al., Classification Trees for Partition Testing, Software Testing, Verification & Reliability. 3, Nr. 2, 1993

[14] Ian J. Goodfellow et. al., Generative Adversarial Nets, arXiv:1406.2661v1

[15] Arniban Charkroboty et. al., Adversarial Attacks and Defences: A Survey Xiv:1810.00069v1

[16] Steidl, Monika, Ruth Breu, und Benedikt Hupfauf. 2020. „Challenges in Testing Big Data Systems: An Exploratory Survey“. In Software Quality: Quality Intelligence in Software and Systems Engineering, herausgegeben von Dietmar Winkler, Stefan Biffl, Daniel Mendez, und Johannes Bergsmann, 371:13–27. Lecture Notes in Business Information Processing. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-35510-4_2.

[17] Felderer, Michael, Barbara Russo, und Florian Auer. 2019. „On Testing Data-Intensive Software Systems“. arXiv:1903.09413 [cs], April. http://arxiv.org/abs/1903.09413.

[18] English, L.P. Improving Data Warehouse and Business Information Quality: Methods for Reducing Costs and Increasing Profits; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 1999.

[19] Wang, Richard Y., und Diane M. Strong. 1996. „Beyond Accuracy: What Data Quality Means to Data Consumers“. Journal of Management Information Systems 12 (4): 5–33. https://doi.org/10.1080/07421222.1996.11518099.

[20] Huang, Xiaowei, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, und Xinping Yi. 2020. „A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability“. arXiv:1812.08342 [cs], Mai. http://arxiv.org/abs/1812.08342.

[21] Kim, Jinhan, Robert Feldt, und Shin Yoo. 2018. „Guiding Deep Learning System Testing using Surprise Adequacy“. arXiv:1808.08444 [cs], August. http://arxiv.org/abs/1808.08444.

[22] Ma, Lei, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, u. a. 2018. „DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems“. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 120–31. Montpellier France: ACM. https://doi.org/10.1145/3238147.3238202.

[23] Pei, Kexin, Yinzhi Cao, Junfeng Yang, und Suman Jana. 2017. „DeepXplore: Automated Whitebox Testing of Deep Learning Systems“. In Proceedings of the 26th Symposium on Operating Systems Principles, 1–18. Shanghai China: ACM. https://doi.org/10.1145/3132747.3132785.

[24] Sun, Youcheng, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, und Rob Ashmore. 2019. „Structural Test Coverage Criteria for Deep Neural Networks“. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 320–21. Montreal, QC, Canada: IEEE. https://doi.org/10.1109/ICSE-Companion.2019.00134.

[25] Wicker, Matthew, Xiaowei Huang, und Marta Kwiatkowska. 2018. „Feature-Guided Black-Box Safety Testing of Deep Neural Networks“. arXiv:1710.07859 [cs], Februar. http://arxiv.org/abs/1710.07859.

[26] Cheng, Chih-Hong, Georg Nührenberg, Chung-Hao Huang, Harald Ruess, und Hirotoshi Yasuoka. 2018. „Towards Dependability Metrics for Neural Networks“. arXiv:1806.02338 [cs, stat], Juni. http://arxiv.org/abs/1806.02338.

[27] Dong, Yizhen, Peixin Zhang, Jingyi Wang, Shuang Liu, Jun Sun, Jianye Hao, Xinyu Wang, Li Wang, Jin Song Dong, und Dai Ting. 2019. „There is Limited Correlation between Coverage and Robustness for Deep Neural Networks“. arXiv:1911.05904 [cs, stat], November. http://arxiv.org/abs/1911.05904.

[28] Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. (2017, July). Safety verification of deep neural networks. In International Conference on Computer Aided Verification (pp. 3-29). Springer, Cham. (e.g. Huang et al., 2017; Ehlers, 2017; Cheng et al., 2017; Tjeng et al., 2018)

[29] Ribeiro, Marco Tulio, Sameer Singh, und Carlos Guestrin. 2016. „‚Why Should I Trust You?‘: Explaining the Predictions of Any Classifier“. arXiv:1602.04938 [cs, stat], August. http://arxiv.org/abs/1602.04938.

[30] Bansal, Aayush, Ali Farhadi, und Devi Parikh. 2014. „Towards Transparent Systems: Semantic Characterization of Failure Modes“. In Computer Vision – ECCV 2014, herausgegeben von David Fleet, Tomas Pajdla, Bernt Schiele, und Tinne Tuytelaars, 8694:366–81. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-10599-4_24.

[31] Miller, Tim. 2018. „Explanation in Artificial Intelligence: Insights from the Social

[32] Chen, T. Y., S. C. Cheung, und S. M. Yiu. 2020. „Metamorphic Testing: A New Approach for Generating Next Test Cases“. https://doi.org/10.48550/ARXIV.2002.12543.