https://doi.org/10.1142/9789811239922_0009

Chapter 9

On the Application of Machine Learning in Software Testing

Nour Chetouane* and Franz Wotawa†

CD Lab for Quality Assurance Methodologies for Cyber-Physical Systems Institute for Software Technology, Graz University of Technology, Graz, Austria

**nchetoua@ist.tugraz.at*, †*wotawa@ist.tugraz.at*

9.1 Introduction

Software is running almost everywhere providing the backbone of our society enabling communication between persons almost everywhere on earth, allowing to come up with automated and autonomous driving function aiming at increasing safety and comfort, or providing fast and reliable simulations necessary for drug development. Without software many businesses would not exist anymore in the current form including social networks or search engines allowing a fast access to information for everyone. Considering the still growing importance of software, it is of uttermost importance to assure that the software running is dependable, i.e., trustworthy, safe, secure, reliable, maintainable, and available.

In order to assure that software and systems meet such requirements like dependability, we need to come up with measures allowing to verify these requirements. Because of the growing size of software, its interaction and integration with other software and systems, formally proving meeting given requirements seems to be not a viable solution. Hence, there is a strong need for methods and techniques that allow at least to indicate whether the developed software meets its requirements. Testing, i.e., executing the software considering well-specified prerequisites and checking for conforming to given expectations, provides such means. Although, testing cannot be used for proving that a given software works correctly, testing provides evidence that this is the case of course depending on the way testing is carried out.

Software testing is an inevitable activity in any current software development process in order to assure quality but requires, in addition, a lot of effort for developing tests and also for carrying them out. In order to reduce testing effort but still providing enough evidence that the test assure meeting given quality criteria, we need to further automate testing. Currently, the degree of automation of test execution in practice is high and still increasing. There are a lot of frameworks helping to carry out tests automatically. However, there are still certain limitations. In case of test generation the situation in practice is different. Most of used test cases have been manually generated and maintained. Therefore, there is a need for providing methods that support test automation.

Machine learning has proved to be successfully applied in many areas like prediction and also classification. Autonomous driving without identifying objects from images often relying on machine learning approaches like deep neural networks, would not have been possible. Hence, there is the question whether machine learning can also be applied in the context of software testing. In this book chapter, we discuss several but not all applications of machine learning to all different aspects of software testing including fault prediction, test case generation, and test suite reduction. In all these application domains, machine learning provides valuable results often outperforming results obtained using other methods and techniques.

We organize this chapter as follows: We first briefly introduce background information regarding testing and machine learning in Sec. 9.2. In Sec. 9.3 we discuss the different research methods and techniques that have been proposed for solving certain testing tasks. The objective behind this section is to provide an overview of the application of machine learning to software testing allowing students, researchers, and practitioners to quickly obtain information of how certain machine learning methods can be applied to solve a particular software testing task. Finally, we conclude this chapter summarizing the findings.

9.2 Background

Before starting summarizing and discussing research methods and techniques for software testing utilizing means of machine learning, we discuss the basics behind software testing and machine learning. For the basic objectives and ideas behind software testing, we refer the less experienced reader to introductory books on software testing like [1]. For an introduction into machine learning there are many text books available including [2].

9.2.1 *Software testing*

Software Testing is without any doubt one of the most important quality assurance activity to be carried out as part of the software development process. The main objective behind software testing is to reveal faults in programs. This is done via searching for certain input values or sequences of interaction that lead to an unexpected behavior of the program or system. From a general perspective, software testing is the process of evaluating a system (or software) under test (SUT) by observing its execution with the intent of finding deviations between the behavior of the SUT and the expected behavior, which is given accordingly to the SUT's requirements [3].

Edsger W. Dijkstra once mentioned that *testing shows the presence, not the absence of bugs*, which is the main limitation of software testing. Therefore, researchers have been substantially contributing to the research field of software testing to come up with ideas and approaches that allow them to decide when to stop testing having sufficient confidence that the program was thoroughly tested. This has been a motivation for many research studies that resulted in several methods and techniques to serve software testing tasks such as the automation of test oracles, software defect prediction for focusing on the most problematic parts of a program, or test case design.

In the following, we briefly introduce the basic principles behind software testing and categorize this research field, which we use later when discussing the application of machine learning. Software testing basically comprises two parts: (i) test suite generation, and (ii) test suite execution. In the context of testing a test suite that is a set of test cases, where a test case is more or less a specification of inputs or sequences of interaction with the SUT together with a test oracle. The test oracle is a means for deciding whether the SUT's output or behavior deviates from the expected ones, when executing the SUT using the inputs or given interactions. A test oracle can range from a set of expected output values to checking for crashes occurring during execution. A test oracle is basically a mechanism for determining whether a test failed or passed.

Both parts of testing, i.e., generation as well as execution, can be at least partially automated. For test suite generation, there exists various methods and corresponding tools like model-based testing [4], combinatorial testing [5], or random or fuzz testing [6] to mentioned some of them. For test suite execution there are tools available like JUnit for programs written in Java. However, despite the increasing use of software-test, automation software testing is still considered to be the most challenging and expensive part of software development.

In order to perform sufficient testing during the many changing stages of a software's lifetime, it is required to have an effective test suite as it presents a major factor contributing to the adequacy and facility of the testing process with regard to time and cost as well. Thus, it is significantly important to evaluate test cases quality in terms of their fault detection capability, which gets more challenging as the level of automation increases. In addition, highly automated systems usually require large test suites, which as well tend to grow in size when the software evolves making the execution of the entire test suites very time-consuming and expensive to conduct. In this regard, many research activities in the software testing field have been proposed with the purpose of optimization of test cases generation, test suite reduction, test cases evaluation and prioritization. Several of the proposed approaches, dealing with these topics, have revealed that the use of Machine learning algorithms can be very effective for solving such issues and improving the testing quality in general. For describing the application of machine learning in software testing we make use of the following categories:

- *Software fault prediction* deals with estimating the fault proneness of programs and their parts for gaining information regarding code quality or other aspects that can be used for focusing testing efforts. In this context, someone would spend more time or money in those parts that are more likely to cause a failure.
- *Test oracle automation* captures all activities to automate test oracles or its generation.
- *Test case generation* is the most important challenge of testing comprising methods, techniques and tools for the generation of test suites focusing on failure revealing capacities of tests.
- *Test suite reduction, prioritization and evaluation* are necessary activities to keep the number of tests to be executed low without compromising failure revealing capacities. The objective is to find faults as early as possible not exhausting available budget or time restrictions.

- Under the category *other tasks* we summarize all other activities and tasks to be carried out during testing not captured by the previously introduced categories.

9.2.2 *Machine Learning*

Machine learning (ML) [2] aims at automatically generating models from available data more or less replicating learning processes we see in nature. Like other models the expectation is to use the learned models for understanding principles, doing reasoning, and making predictions using observations. Moreover, when being able to learn from data we may also gain a deeper understanding of the human brain. From a general perspective, ML algorithms are centered around learning a function that maps an input domain (i.e., data points) to an output domain with the goal of achieving a certain task, e.g., making a certain decision in a given situation, or predicting figures like the costs of shares at a stock exchange. The underlying idea behind ML is to allow programs to improve their performance and prediction accuracy mainly through experience, i.e., all the data that becomes available over time. The use of ML is rather appealing because it would allow to obtain models directly from data in an automated way not requiring to handcraft such models.

Over the past decades, ML has proven to be of great practical value for many application. Actually, ML has become a common tool in the majority of tasks requiring information extraction from large data sets. ML algorithms, have found their way, as well, into software development practice as these algorithms offer viable solutions for many software engineering issues. In this chapter of the book, we focus in particular on their major role in serving various purposes in the area of software testing such as detecting potential errors, constructing test oracles, or generating test cases (see [7]).

The two canonical types of ML settings are supervised and unsupervised learning. Supervised learning describes a scenario in which the training data contains significant information. Typically, training examples are provided with labels which are missing in the unseen test data to which the learned model is to be applied. Two classic supervised ML tasks are regression and classification. In unsupervised learning, however, no previous information about the training examples is available, therefore, there is no distinction between training and test data. The learner processes unlabelled input data with the goal of identifying patterns or discovering similarities that can lead to some useful analysis, or come up with a compacted version of that data. The most prominent example of such a task is data clustering where a set of data is partitioned into subsets of similar objects [8].

In ML, a solution can be essentially created in two ways. Either a learned model is constructed from the data directly, which is used in decision trees induction, or it is obtained using search methods that look for an effective solution among a set of candidate solutions. For example, in the domain of neural networks gradient-descent algorithms are used to search for the neuron weights. Stochastic search algorithms such as genetic algorithms (GA) are often used in machine learning applications. They are considered, i.e., in reinforcement learning algorithms where the learning system performance is evaluated by a fitness metric. Hence, they are effective in situations when the only available information is a measurement of performance (see [9]).

In software testing a lot of different ML algorithms have been already used such as decision trees (DT), artificial neural networks (ANN), Bayesian networks (BayesNet), clustering (e.g., K-means), or support vector machines (SVM). Depending on the task one or another ML method might be the most appropriate one requiring to carry out experimental evaluations. The specific task and as well the setup we are facing also influence the choice of the ML method to be used. In case we are interested in prediction or classification, we may make use of supervised ML methods. If we are interested in investigations regarding certain patterns, we may rely on unsupervised methods like clustering.

9.3 Applications of Machine Learning in software testing

In this section, we discuss selected research papers employing the most common ML algorithms like ANN, DT, SVM, k-means, and others for carrying out software testing tasks such as fault prediction, automating test oracles, designing test cases, evaluating test suites, or reducing and prioritizing test cases. Most of the ML methods are designed to learn a mapping function from input data to outputs in order to make predictions or classifications. Therefore, we describe the majority of selected approaches in terms of inputs that are fed into the ML algorithms, and the outputs that are delivered after the performing learning.

9.3.1 *Machine Learning for software fault prediction*

Software fault prediction is a very useful and important task to estimate the expected delivered quality and maintenance effort before the deployment of software. The main goal of software fault prediction is to track and reduce the number of latent software defects as early as possible in the software life cycle, which allows improving the effective software costs, the reliability of the software to be developed, and the achieved customers' satisfaction (see [7]). In the following, we provide a brief discussion of research work in the category of the application of ML for software fault prediction depicted in Table 9.1.

Table 9.1 Research papers describing the use of ML methods applied for software fault prediction.

Papers	**ML method used**
[10]	k-means clustering
[11]	GA
[12]	Naive Bayesian Network, SVM
[13]	DTR
[14]	ANN
[15]	ANN
[16]	GA
[17]	ANN
[18]	ANN
[19]	Ensemble Algorithm
[20]	Ensemble Algorithm
[21]	ANN, unsupervised Self-Organizing Map (SOM)

[10] showed that unsupervised techniques like clustering can be used for software fault prediction especially when fault labels are not available. In this paper, K-means is used for predicting faults in program modules. Cluster centers were initialized by a quad trees based method. The dimensions and metrics used in the data sets represent several metrics thresholds (i.e. Lines of Code (LoC), Cyclomatic Complexity (CC), Unique Operator (UOp), etc. A threshold vector was used for cluster evaluation. As for each cluster, if any metric value of the centroid data point is greater than the threshold, then, that cluster is labeled as faulty, otherwise it is labeled as non-faulty. After evaluation, the overall error rates of this approach are found to be better in most of the cases, than other existing algorithms using other k-means initialization methods.

GA was applied in [11] to determine most important attributes for predicting faulty modules. Raw data was collected in the form of structured source code of open source systems. The collected data is evaluated using some of Chidamber and Kemerer (CK) software metrics such as LOC, Weighted Methods per Class (WMC), Response for a class (RFC), etc. After data filtering, the metric values are collected and converted into a binary form (i.e. 0, 1) based on a specific threshold. The transformed values are given afterwards as inputs to the GA. After applying mutation and crossover steps on the given binary inputs, the GA outputs a string of binary values showing the most important attributes corresponding to mostly fault prone metrics (i.e. having the value 1). Then, a detailed analysis of the algorithm performance, is made based on a confusion matrix of the GA prediction outcomes.

GAs were also used in [16] for predicting software errors. In order to forecast an error in an application, the proposed method goes through three main steps: first selection of an appropriate software database to use software indices. Second, application of the GA in order to extract important features and then make use of the GA output to determine the probability of an error in the application. Results showed a good performance of the suggested method in terms of time required for predicting errors, and an error detection rate reaching 95%.

In order to improve the quality and reliability of software, many authors suggested the use of ANNs. In [14], authors developed and trained a software defect prediction model using a feed forward back propagation network, with the goal of reducing the overall cost and development time. The data set used for training the network comprises nine software metrics attributes measured for each phase of twenty genuine software projects such as; requirement stability (RS), design review effectiveness (DRE), process maturity (PM), etc. The ANN takes these software measurements as inputs and predicts four density defect indicators for each phase (i.e. requirement analysis phase, design phase, code phase and testing phase). After the training phase, a predictive analysis of software defects and their threshold is done. The testing results showed the proposed approach to be good and effective as the resulting model was able to detect software defects with very less error rate. In their paper, [15], also train an ANN to identify the error prone parts of the software and predict whether a module is erroneous or not. The ANN has been trained for classifying software modules into groups of faulty and non faulty modules. The dataset used for training and testing the model also represents different software metrics attributes. The experimental results showed that the approach provided a good fit in terms of less error in prediction comparing to existing analytical models.

Similarly, [17] introduced an ANN that solves the software fault prediction as a binary classification problem that determines the faulty or fault-free status of a software module. Also, [18] presented a non-linear hybrid supervised learning method combining ANN with gradual relational association rule mining for classifying defective and non-defective software entities. As for [21], the authors focused on determining the relationship between object-oriented metrics and fault proneness at the source code class level. A hybrid model was designed, as well, combining an unsupervised SOM algorithm and a multi-layer ANN for prediction of faults occurrence. Six CK metrics have been used as input nodes to the ANN and a prediction rate as the achieved output which refers to a pair of fault and fault free classes. The presented validation results in the above studies proved the associativity of software metrics to the fault proneness.

In [13], the authors investigated the capabilities of DT Regression (DTR) for the prediction of the number of faults in given software modules under two different scenarios; intra-release prediction and inter-releases prediction. For the intra-release prediction scenario, the authors performed a 10-fold cross validation for training and testing the model. For the second scenario, previous software releases were used to build the model. For testing it, authors made use of the current release of the same software. DTR is build using independent variables that are software metrics to predict a numeric outcome which is the number of faults in a software module. The experimental study was carried out relying on fault datasets corresponding to five open-source software projects comprising multiples releases. The results showed that DTR provided significant accuracy for the number of faults prediction across all datasets in both considered scenarios.

[19] applied ensemble classification learners to develop a model for predicting fault proneness. The authors proposed a framework for validating source code metrics using the performance of the classifiers to evaluate each candidate metrics and select the right set of metrics that improve the performance of the fault prediction model. The author combined five classification techniques (i.e. one Logistic regression and four different types of ANNs), then computed their final output. The classification models are trained on a data set including twenty source code metrics and module fault information corresponding to several software projects. The experiment proved that multiple combined learners perform better than independent single model in terms of prediction accuracy and lower costs. Later, in [20], the authors, in addition, proposed linear homogeneous ensemble methods using an extreme learning model consisting of an ensemble of single-hidden layer feed forward NNs which combine all their predictions to get a final fault prediction result.

[12] proposed a feature selection technique that is applied for classification-based fault prediction. The authors suggested to train a probabilistic Naive Bayes model and an SVM classifier on historical data of software. The trained models are used afterwards for predicting faults in software. In this study, the authors make use of two public available data sources; from the NASA IV&V facility and from open source Eclipse Foundation, representing a set of non static code features. Each observation in the data sets represents a software module. It consists of an ID, several static code features such as McCabe, Halstead metrics, and LOC. Plus, an error count which, when equal to 0, indicates that no errors are recorded for this software module, and otherwise it takes 1. The data used for training and validating the models is first selected, then, static code metrics are computed on the software source code, and then saved in an attribute relationship file format (ARFF) which is used as a test set for the models.

9.3.2 *Machine Learning for test oracles automation*

Software testing involves executing a program under test and examining the output of the program whether it conforms with the expectations or not. A test oracle is a mechanism required in functional testing to determine whether a test failed or passed when being executed. The oracle can be a human or a piece of software that often operates separately from the system under test (see [7]). Note that the oracle problem, i.e., finding an appropriate test oracle, is still an important topic of testing. In the following, we discuss papers given in Table 9.2 where ML is used for obtaining a test oracle.

Table 9.2 Papers dealing with the application of ML methods for test oracle automation.

Papers	**ML algorithms used**
[22]	DT(C4.5 algorithm)
[23]	GA, SVM
[24]	ANN
[25]	ANN, DT
[26]	ANN
[27]	ANN
[28]	Knowledge Discovery in Database, AdaBoost, JRip

[23] proposed a novel approach to evolve a test oracle making use of genetic programming and SVMs. The authors aim to model specifically the behavior of programs which process and output sequences of integers.

An input/output list relation language (IOLRL) was designed to formally describe the relations between the input and output lists of these software programs. Plus, a GA was applied to evolve relations in IOLRL that can well distinct passing and failing test cases and encode test cases into bit patterns. These bit patterns and some labelled test cases are employed to train an SVM classifier which is used as a test oracle to verify software behavior.

[24] showed that ANNs can be used for constructing a test oracle to automatically handle the mapping between the input domain and the output domain. After training an ensemble of ANNs until reaching the adequate error rate. The complete test oracle output vector is composed of the results of all NNs. Two industry-sized case studies are used for the evaluation. A mutated faulty version and a fault-free of each case study are created in order to test the capability of the proposed oracle to find injected faults. Afterwards, actual outputs produced by the SUT are decided to be faulty or not compared to the ANNs output using an automated comparator. The results proved this multi-networks oracles detected up to 98% of the injected faults with an accuracy of 98.93%.

A similar approach for black box testing using ANN with a comparison tool was proposed by [26]. The network was trained on randomly generated data and was able to classify the test data with 100% accuracy and basically becomes a simulated model of the software application. When executing the software on real data, the network was able to monitor its behavior. This trained network is assumed to be fault-free. In the evaluation, some faults are inserted to the tested program, and a tool is used to compare the SUT output to the network output and decide if is correct or not. The results of experiments on one of the benchmarks programs showed that 92% of faults were detected by the neural network.

[27] presented a similar methodology for testing real time applications using a Back-propagation ANN. Training input data is generated randomly and then fed to the network. Changes with mutation testing are applied to generate faulty versions of the original program. For each input, a comparison tool is used to evaluate the correctness of the obtained results based on the absolute difference between the ANN output and the corresponding value of the program output. If a large difference is recorded between the two outputs then a program defect is indicated. Back-Propagation learning method ensures that the trained network can be updated by learning new data for evolving versions of the software. However, the author reclaims that this approach could have some limitations in case of larger combinations of inputs and outputs.

[25] provided a method for learning a test oracle utilizing ANN and DT model to generate expected outputs. The DT model can also be used for detecting software faults along with mutation testing. In the evaluation, the authors trained and tested the two models on the well-known Triangle program. The study showed that DTs provide maximum accuracy as a data mining technique for test oracle however the proposed DT based approach can only be used for small programs which take integer input data besides the framework is not reliable enough for non-deterministic software. Similarly, in [22], a test oracle based on DT was proposed for identifying failures using binary classification model in order to improve the testing of software with graphical interfaces such as Mesh simplification. They train a C4.5 DT classifier on available known samples, then let it label unseen test cases produced from other programs.

[28] proposed an approach based on ML for automating test oracle mechanism in software. In this paper, a knowledge discovery process is applied on historical usage data in order to define the topology in relation to the inputs and outputs of the ML algorithm. After the selection of two convenient ML algorithms (i.e. AdaBoost, JRip) and their hyper-parameters, each of them is trained to generate an oracle suitable for the SUT. The trained model is then used to classify each of the test cases to be run on the target SUT in one of three classes: “Valid”, “Fault” and “Possible Fault”. To evaluate this approach, the authors carried out three experiments on a web application; first using randomly inserted failures, second using failures inserted by an expert and thirdly using mutation testing. The results showed an accuracy of 94%, 72% and 98% respectively. However, some limitations related to the generalization of the proposed approach have occurred since the example application used in this experiment has a limited set of functionalities.

9.3.3 *Machine learning for test cases generation*

Test case generation is the process of designing test suites, which is one of the most important parts of software testing aiming at providing test suites having a high fault detection rate. A test suite comprises test cases describing inputs and interactions with the system together with the expected output or behavior of the system. A test case can either pass or fail. It fails if the output or behavior deviates from the expected output or behavior when executing the program using the given inputs or interactions. Otherwise, it passes. In Table 9.3 we list some ML applications used

Table 9.3 Papers introducing ML methods applied for test cases generation.

Papers	**ML algorithms used**
[29]	ANN, GA
[30]	GA
[31]	GA
[32]	GA
[33]	GA
[34]	GA
[35]	ANN, GA

for test case generation, we are going to further discuss. It is worth noting that we also added papers that utilize GA without ML in this section for the sake of completeness.

[29] proposed a novel approach for designing adequate test cases on the basis of software specifications. The approach focuses on creating test cases from output domain instead of the input domain. A neural network was trained to be taken as a function substitute for the SUT with the same number of inputs and outputs. The initial inputs of the NN are generated randomly and its outputs are considered as the actual outputs according to the SUT specifications. Based on the function model, the GA searches for the correspondent test inputs for a given output, applying a series of operations (i.e. reproduction, crossover and mutation). The GA stops when its fitness function reaches the maximum value meaning that the corresponding test inputs have been found. Experiments conducted on two different software programs showed that this proposed approach is promising and effective.

[30] made use of GA for fully automating test case design with more focusing on boundary value analysis tests. GA searches for effective test cases in the input domain of the SUT. The initial population of test cases is generated randomly, the Fitness function used here is measured as the difference of each test case from the boundaries of the variable. For selection criteria, the GA uses the roulette wheel method with respect to the probability distribution based on fitness value. The authors compared their approach to Random Testing (RT) and they observed that not only GA outperformances RT but also the overall quality of software is improved in comparison to using RT. The study confirms that GAs are quite useful for increasing the efficiency and effectiveness of software testing, and hence to decrease the overall development cost for software-based systems.

[31] focused on black-box testing methods such as RT and adaptive RT, aiming to optimize a specific string test case generation. A multi-objective optimization algorithm based on GA is used for generating a diverse and effective set of test cases. Several string distance functions were introduced to compute the length distribution of the string test cases, and for controlling the diversity of test cases within a test set as well. An empirical study was performed on several real-world programs. The results have shown that the generated string test cases outperform test cases generated by other methods.

[32] made use of GA for automatic test cases generation and optimization. The proposed algorithm starts by randomly generating initial input tests and executing them. Only test cases whose path coverage is more than 20% are selected as initial population for the GA. New tests are generated after applying GA operations (Crossover and Mutation). For checking the coverage criteria, a probabilistic based fitness function that uses 0.8 probability for crossover and 0.2 for mutation operation. The algorithm stops when path coverage including statement and branch coverage exceed 95%. This method, based on an optimized fitness function, helps achieve final test suites with 100% path coverage.

[33] focused on GA for optimization because it was successfully used by many researchers for providing a more feasible and reliable test suites. This study proposed an optimization approach for test case generation combining GA and mutation testing. For evaluating the performance of the test cases, the authors made use of the mutation score as a selection criteria for the newly produced test cases. The fitness function of the underlying GA relies on the capability of a test case to kill mutants.

[34] focused on structural testing at the unit level. It presents a framework of test case generation using an improved adaptive GA. The main challenge is to search for a set of test cases that lead to the highest path coverage. The proposed method improves search efficiency by maintaining a high population diversity by dynamically adapting crossover rate and mutation rate. Experimental evaluation of the proposed framework were performed making use of six industrial programs. The results approve that the proposed method is more efficient than existing similar methods and also random generation of test cases for assuring path coverage.

[35] proposed a neural network based test case generation approach for data-flow oriented testing. The authors focused on the data flow testing criterion as coverage objective for test case generation. Therefore every variable definition in the program and its use (DU-pairs) need to be

calculated first. A Back Propagation NN is then trained to simulate a fitness function that is mainly based on the control flow graph of the program. The fitness function checks if the test case covers the given DU-pair. Afterwards, a GA is used to generate test cases to cover all DU-pairs. Thus, the fitness value of the test case can be evaluated with the NN instead of running the instrumented program. The results showed that this approach reduces the total time of test case generation especially in case of large and complex program, when compared with traditional GA based methods.

9.3.4 *Machine learning for test suite reduction, prioritization and evaluation*

In this subsection, we discuss the application of ML for test suite reduction, prioritization and evaluation taking care of the papers given in Table 9.4.

Table 9.4 Some research studies of ML methods applied for test cases reduction, prioritization and evaluation.

Papers	ST tasks	ML method used
[36]	Test cases generation, test cases reduction	ANN
[37]	Test cases reduction	Hierarchical clustering
[38]	Test cases reduction	Cluster analysis
[39]	Test cases reduction	semi-supervised clustering with k-means
[40]	Test case prioritization and selection	K-means, Expectation-Maximization (EM), Incremental Conceptual Clustering (Cobweb), DT
[41]	Test cases reduction	K-means clustering
[42]	Test cases reduction	K-means clustering
[43]	Test cases generation, test suite evaluation	DT for model inference, GA
[44]	Test cases reduction	DT for model inference
[45]	Test cases prioritization	ANN
[46]	Test cases reduction	K-means

When testing programs in practice the question whether the used test suite is effective enough arises. This also tells when to stop testing. So, it is significant to evaluate the quality of the underlying test suite in terms of fault detection for a specific program or SUT. Recently test suite evaluation has become a major focus in software quality assurance research.

For assessing the quality of a test suite, the highest number of existing approaches have focused on the source code level of a system assuming that adequacy of test suites is measured by its ability to execute all statements, branches, or mutants. Yet, in practice, such syntax based criteria of testing are considered, at best, as minimum requirements for a test set and they are rarely sufficient, and, sometimes misleading even when they are satisfied. An alternative approach has been proposed by [43] in order to overcome the shortcomings of syntax based adequacy metrics, the authors recommend the idea of behavioral coverage which basically consists in inferring a suitable model that reflects the system behavior during the execution of its test cases and can be used to provide a much more reliable assessment of test suites. If the inferred model correctly covers the behavior of a system and is accurate enough, then the test suite can be considered to be adequate. The paper allows the use of ML algorithms such as DT for model inference, it also presents a search-based test generation technique using GA which extends standard syntactic and optimizes the generation of rigorous test sets. The authors used a selection of Java units for the empirical evaluation, the results demonstrate that test cases with higher behavioral coverage significantly outperform current baseline test metrics in terms of detected faults.

Test suite size has a direct impact on the cost and effort of software testing. In this regard several different approaches have been proposed for test suite reduction including test suite minimization, test case selection and test cases prioritization. Test suite minimization is a process of detecting and then removing the obsolete redundant test cases from the test suite. Test case selection consists in choosing a representative subset of test cases from the original test suite that will be used to test the newly introduced parts of the software. Test case prioritization focuses on the identification of the ideal ordering of test cases that maximize desirable test requirements, such as early fault detection. These test suite reduction techniques must preserve the highest possible test suite fault detection capability in addition to minimize the size and the time of the testing process.

[36] presented an automated approach for test cases reduction, mainly based on input-output analysis of the tested software. First, a multi-layer ANN is build with similar characteristics as the SUT such as types and number of inputs and outputs. For obtaining the ANN training set, input data is generated randomly and then fed to the original program which generates the corresponding outputs. During the ANN back-propagation training process, a penalty function is used in order to assign lower weights to less important connections. Then, ANN is pruned by removing edges which have lower weights without affecting the predictive accuracy of the network. Thus, only connections corresponding to the most important

attributes are retained in the network. Then feature ranking is performed by sorting inputs in the order of their importance; according to the set of weights from the pruned algorithm. This helps to identify the most significant attributes contributing to the value of output. Test cases with lower ranked attributes are thus removed. After pruning, rule extraction phase is performed in order to express the Input-Output relationship in the form of If-Then rules. Then, clustering is applied to build equivalence classes for continuous attributes. Afterwards, test cases are generated by making combinations of data values of the inputs. Hence, the reduction of size of the input domain conducts to the reduction in test cases. ANN was also applied as black box testing method for test cases prioritization in [45]. Input data is randomly generated by extracting inputs from design specifications and correspondent outputs are obtained by manually executing the input tests. The ANN is trained to assign priorities to the obtained test cases based on a set of priority assignment rules using design specifications and software requirement specifications. The network output is the priority assigned to corresponding test cases. Experiments have been performed on different NN with 2 to 20 layers, the results show that the NN can be used for effectively predicting a test case priority.

Clustering techniques were shown by several research studies to be very useful for test cases reduction purposes. In [37], authors proposed a clustering based approach to help making a selection of diverse test cases and removing redundancy. The authors introduced a framework which provides a group of test cases comparison metrics which quantitatively compare any random pair of test cases. Besides, a hierarchical clustering algorithm is performed using program profiles and static execution in order to group similar test cases. A specific threshold is used to drive redundancy elimination, test selection and effectiveness of new test cases. The experimental results identified 10–20% of redundant test cases.

In [38] the authors provided a solution to deal with the trade-offs between test suite reduction and fault detection capability in regression test selection. Clustering of test cases execution profiles was performed to group program executions that have similar features which helps to better understand program behaviors so that test cases can be properly selected to reduce the test suite effectively. The results showed that this approach can significantly reduce the size of test suite, on the premise of finding most of fault-revealing test cases.

Later, authors, in [42], also performed a cluster analysis on other different types of structural profiles which consider sequential, relations and structural information between function calls including function execution sequence, function call sequence and function call tree reduction. The reason behind this study is that authors believe that binary or numeric vector-based methods which consider only the number of times that a function or statement is executed do not always generate satisfying results. K-means cluster analysis was also applied to select a representative subset from the original test suite based on the similarity of profiles of exercised tests.

In [39], semi-supervised clustering was used for the first time to improve clustering results and test selection. Semi-supervised clustering technique aims to derive appropriate information by providing some labelled data in form of constraints. In this paper, the authors introduce a semi-supervised K-means where a limited supervision was provided in the form of constraints derived from previous test results and recorded execution profiles of tests. For each test, a simple execution profile and a function call profile are generated indicating if the corresponding function is called or not during a running test. The experiment results illustrate the effectiveness of this semi-supervised cluster test selection methods.

In [40], both supervised and unsupervised ML algorithms have been applied. In this paper, main goal is to link test results derived from the application of different testing techniques to functional aspects of the program. First, test results are grouped into similar functional clusters which serve as functional equivalence classes. Three famous clustering algorithms k-means, Expectation/Maximization EM and Incremental Conceptual Clustering (Cobweb) are used. After clustering, test inputs are linked to software functional aspects and then can be used for training a DT model, based on C4.5 algorithm, to generate classifiers, in the form of rules, which can serve for many purposes like test cases selection or prioritization. In [41], the authors proposed a mining approach, also based on clustering, to provide better set of test cases. First, a control flow graph is inferred from the tested program, then, independent paths are clustered using k-means algorithm. A reduced test suite is build including test cases represented by each cluster. This approach eliminates test cases which cover similar testing paths of the program.

In [46] the authors proposed a clustering based approach for test suite reduction using k-means algorithm. Representative test cases are selected from each cluster to build a reduced test suite. K-means was combined with binary search in order to look for an appropriate K number of centroids. For evaluating the reduced test suite, the authors compute branch, statement, MC/DC (Modified Condition/Decision Coverage) coverage and mutation score. The proposed approach can be applied on programs with different types of inputs and outputs (numeric, strings and booleans). A first evaluation have shown an average reduction of 95.9% with same branch coverage as originally but with a small decrease in the mutation score for some examples. Nevertheless, 82.2% reduction was provided with a guarantee of same coverage and mutation score as the original test suite.

DTs were employed by [44] to infer a model from the SUT. The proposed model-based test suite reduction approach was conducted without the need to execute the program. The idea behind is to remove test cases that do not change the learned model by checking the similarity between deduced DTs after each test case removal. This approach maintains almost same level of code coverage and mutation score as the initial test suite.

9.3.5 *Other tasks*

In the following two subsections, we briefly summarize the content of the papers given in Table 9.5, which cannot be assigned to the other software testing categories.

Table 9.5 ML methods and techniques applied in other software testing tasks.

Papers	Other tasks	ML algorithms used
[47]	Software quality evaluation	ANN
[48]	Software performance prediction	ANN
[49]	Software reliability prediction	ANN
[50]	Software reliability prediction	ANN
[51]	Assessing Software reliability	GA, SVM, BayesNet.
[52]	Test time prediction, test cost estimation	Classification tree
[53]	Cost and Execution effort estimation of testing	ANN, SVM
[54]	Estimation of testing effort by predicting test code size	k-nearest neighbors algorithm (k-NN), BayesNet, DT (C4.5), Random Forest, and ANN

9.3.5.1 *Software Quality Prediction*

Software quality prediction models aim to ensure the reliability of the delivered software and help building products of highest quality by predicting quality factors such as whether components are fault-prone or not. Effective verification and accurate prediction of fault prone modules have a major role of eventually increasing productivity and reducing risk.

[47] introduced an approach for software quality evaluation. The approach is mainly based on an improved back-propagation neural network, which is mainly used for building a comprehensive evaluation model for software quality. The inputs of the neural network are evaluation indicators that reflect different levels of software quality. After training the ANN a knowledge base is iteratively formed that can be used for evaluating the software quality comprehensively.

[48] used ANN for predicting the performance of a software by describing the input-output relationship. The authors proposed to make use of ANNs to learn the correlation between the most effective input factors and the software performance.

The two papers [49] and [50] use ANNs for software reliability prediction. The main idea is to use ANN based models to produce accurate prediction results of software reliability based on fault history data without any further assumptions. The proposed neural network predicts software reliability by learning the dynamic temporal patterns of the fault data.

In [51] the authors try to improve software reliability by accurately predicting errors in short time and low cost. For this, the authors suggest to apply a GA to optimize test data and generate effective test cases. In addition, the authors also make use of other machine learning algorithms such as SVM or Bayesian networks to design models for predicting faults in programs at early stages and to repair them. The discussed results show that software reliability fault prediction depends on the number of defects, which are already present in the software before deployment.

9.3.5.2 *Test cost estimation*

Test cost and execution effort estimation is very much important when managing software projects. The aim of test cost estimation is to provide an accurate prediction of the effort needed to develop and test software systems with respect to time and cost required to complete a defined task in the testing cycle.

In [52], the authors made an experiments in order to investigate whether machine learning techniques can be used to determine important software testing attributes and for predicting testing cost and specifically testing time. They create a classification DT in order to identify relevant factors that affect testing time. After analyzing the DT, they noticed that each node represents a group of programs having similar attributes sharing an average testing time. Therefore, they conclude that classification tree can be effectively used in estimating the testing time of a new program as well.

In [53], the authors introduced the application of machine learning algorithms for estimation the execution effort of functional testing. In the paper, the authors focussed on SVM to solve non-linear regression problems. In addition, the authors also trained a feedforward multi-layer perceptron network using an asymmetric cost function that helped to build a model that favors overestimation over underestimation.

In [54] different machine learnings techniques such as linear regression, k-NN, Naive Bayes, DT (C4.5), Random Forest, and ANNs were empirically investigated for performing an early prediction of the effort required to test object-oriented software from the perspective of test code size, i.e., the required test lines of code (TLOC). For training the prediction models, the authors used functional requirements, describing use cases because they are the main available input at an early stages of the software development lifecycle. The results showed, that this use-case metrics based approach is more accurate in prediction of TLOC compared to the well-known Use Case Points (UCP).

9.4 Conclusions

As pointed out by [7] many learning methods have known practical problems such as overfitting, local minima, or curse of dimensionality caused by either data inadequacy, noisy data, irrelevant attributes in data, or incorrect domain theory. Therefore, the application of ML in practice may not be that easy requiring a deeper understanding of the methods and their limitations. For known tasks and domains in the case of software testing, we discussed several methods and techniques already available. For new challenges in the area of software testing we have to take care of the limitations behind the ML methods and algorithms. One of the few key limitations of ML algorithms is the fact that for their application we have to make sure the availability of relevant data. In addition, the ML algorithms sometimes need to procure a previous domain knowledge in order to be able to deploy and interpret the outcomes of ML correctly. Moreover, when applying ML we have to assure a high quality of data because the quality has a direct impact on the outcome of analytical learning method. It is also worth noting that one of the main disadvantages of relying on standard ML algorithms is that some of them often require a sort of preprocessing such as data cleaning, handling missing or null data values, data normalization, and removing inconsistencies in data.

As noticed, ML algorithms differ in terms of their function as some of them seem to be more suitable for automating certain software testing tasks than others. For example, ANNs have been widely used for solving problems related to test oracle automation, predicting faults, prediction of testing costs and software reliability. Their main advantages are their robustness to errors in training data and their ability of learning complex functions like non-linear and continuous functions. However, they are commonly known as black box systems because interpreting what a neural network has learned is not that straightforward and can hardly be interpreted by users directly. In addition, ANNs are known to have slow training and convergence processes requiring adequate computational resources. Besides this ANNs lead to multiple local minima in error surface and suffer from overfitting (see [7]).

GAs have been used more often for test case generation and optimization. They are recognized to be very much suitable for tasks that require an approximation of complex functions. GAs are most efficient in a search space where little information is provided, and they usually only require an evaluation function rather than the availability of a large set of data. Yet, one of their limitations is that they are computationally expensive and time-consuming especially when the solution space is continuous. In addition, most of approaches using GAs in test case generation, need to run the tested program for each generated test case to evaluate its fitness value which costs a lot and consumes a lot of running time, especially for large scaled programs [35]. We have also noticed the current use for clustering techniques such as k-means in test suite reduction. This might be because clustering is a common solution when no labelled data is available, which is usually quite hard to afford in case of real-world data. [55] stated that it is not always easy to truly take advantage of the benefits of ML algorithms without fully considering their assumptions and implications.

References

[1] G. J. Myers, *The Art of Software Testing*, 2nd edn. John Wiley & Sons, Inc. (2004).
[2] T. Mitchell, *Machine Learning.* McGraw Hill (1997).
[3] P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press (2016).
[4] I. Schieferdecker, Model-based testing, *IEEE Software* **29**, 1, pp. 14–18 (2012).
[5] D. Kuhn, R. Kacker and Y. Lei, *Introduction to Combinatorial Testing*,

https://doi.org/10.1142/9789811239922_0010

Chapter 10

Creating Test Oracles Using Machine Learning Techniques

Rafig Almaghairbe[a] and Marc Roper[b]

[a]*Department of Computer Science, University of Omar Al-Mukhtar, Derna, Libya*
rafig.almaghairbe@omu.edu.ly

[b]*Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK*
marc.roper@strath.ac.uk

10.1 Introduction

Automating aspects of software testing process can be a key factor in reducing the costs of software development projects. Consequently, researchers in software testing have developed ways to automatically generate and execute test cases, as well as maintain and manage test suites. However, the generation of a test oracle (a mechanism to determine whether the output associated with an input is correct or incorrect) is a relatively neglected area of research. Whilst many tools have been developed to generate test inputs [1], few tools exist to build automated test oracles, making the process of checking test outputs primarily human-centred and as a result expensive and possibly error-prone [2].

Proposed approaches to generate test oracles vary from the inexpensive and ineffective (e.g. implicit oracles) to effective but very costly (e.g. specified oracles). Implicit oracles are easy to construct at practically no cost (e.g., the work of Carlos and Michael [3]), and usually perform well with general errors like system crashes, null pointer dereferences or unhandled exceptions, but are not applicable for semantic and complex failures [4]. On the other hand, specified oracles can be obtained from formal specifications, and are effective in revealing failures, but defining and maintaining such specifications is a challenging task and hence such specifications are very rare [4].

In recent years, researchers have tried to strike the balance between these approaches and develop a technique which combines the effectiveness of specified oracle and the cost of an implicit one by using machine learning and data mining approaches, in particular anomaly detection, to automatically identify failing tests. The purpose of this chapter is to re-evaluate and analyse the work on test oracles built using such machine learning techniques, and also to identify the properties of automated/semi-automated test oracles required for them to be practically usable. Researchers in the software testing community can use these properties as criteria to evaluate test oracles based on machine learning techniques.

The chapter has been motivated by the following research questions:

- Question 1:
  - (a) What anomaly detection approaches (machine learning, data mining etc.) have been used to build automated test oracles?
  - (b) Which of the variety of anomaly detection approaches are considered to be the most appropriate for automated test oracles?
  - (c) What is the effectiveness of anomaly detection strategies (classification, clustering etc.) for the creation of automated test oracles?
- Question 2:
  - (a) What data from software systems have been used to build anomaly detection models for automated test oracles?
  - (b) What data from software systems are reported to provide the anomaly detection techniques with the best chance of building an effective automated test oracle?
  - (c) How has this data been transformed to suitable set of feature vectors?
- Question 3:
  - (a) What types of software faults have been used on the empirical studies of automated test oracles by using anomaly detection techniques?
  - (b) What classes of faults are reported to be the most frequently detected via anomaly detection techniques?

(c) Is there any relationship between specific classes of faults and the success of anomaly detection approaches?

The remainder of the chapter is organised as follows: In the next section, a background on test oracles is presented. Section 10.3 provides an overview of the related work. Section 10.4 discusses work that addresses the problem of test oracles creation based on machine learning techniques and answers the research questions identified above. Section 10.5 reports the required properties of test oracle techniques. A road map for further research direction is outlined in Sec. 10.6. Section 10.7 summarises and presents the conclusions.

10.2 Background on Test Oracles

An oracle can be defined as a mechanism that determines and judges whether a system's test results have passed or failed [5]. This function can be carried out by the tester (human oracle), or by automated/semi-automated means. Memon *et al.* [6] defined two important parts of a test oracle: oracle information that represents expected output, and an oracle procedure that compares the oracle information with the actual output. Shahamiri *et al.* [7] summarised the test oracle process as follows: (1) generate expected outputs; (2) save the generated outputs; (3) execute the test cases; (4) compare expected and actual outputs; (5) decide if there is a fault or not. It is worth pointing out that the test case execution activity does not form part of the test oracle, but it is part of the oracle process.

Ye *et al.* [8] have characterised a perfect and complete automated test oracle as follows: (1) it should have source of information which makes it possible to produce a reliable and equivalent behaviour to the software under test (SUT); (2) it should accept all entries for the specified system and always produce the correct result; (3) it should have the answers to the data which is actually used in the test. Again a point worth noting is that the anomaly detection approaches discussed in this chapter are unlikely to meet the first of these criteria.

A traditional and generic test oracle structure can be seen in Fig. 10.1 [9]. In this scenario, the test oracle accesses the set of data required to evaluate the correctness of the test output. The set of data can be obtained from the specification of the SUT and holds enough information to support the oracle's final decision. The following subsections introduce several structures of test oracle functions using different sources of information.

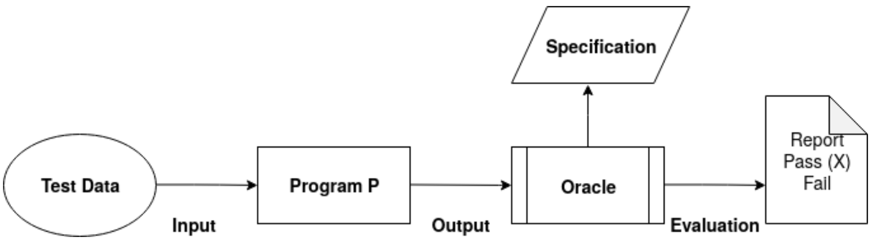

Fig. 10.1 Generic Test Oracle Structure.

10.2.1 *Test Oracles Based on Individual Test Cases*

The development of unit testing frameworks allows the developer to specify the pass/fail conditions for an individual test. This approach is illustrated in Fig. 10.2 which shows how the test case itself carries the expected results in order to decide the correctness of the SUT [10]. The tester can implement this oracle by using one of the various frameworks that are known collectively as an "xUnit" family which support unit testing for a range of programming languages [11] (e.g., "JUnit" is an "xUnit" framework for Java). Testers develop test oracles in their code by inserting assertions in a program to check unit or partial results. This oracle still demands a lot from the developer in that test cases need to be hand coded and acceptable results clearly specified. An example of such oracle is shown in the following piece of code:

```
public void testBOOKInLibrary ( ) {
// A test oracle to check the correctness of the
// method "boolean Library.checkByTitle(String)"
Library library = new Library ( );
boolean search = library.checkByTitle ("Data Mining");
assertEquals (true, search);
}
```

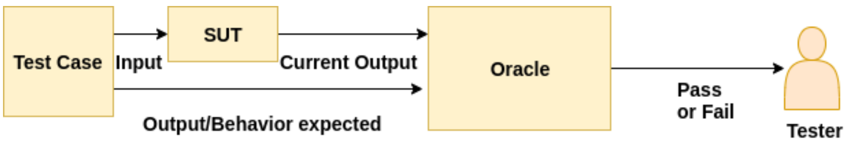

Fig. 10.2 Test Oracle Structure Using Expected Output Behaviours.

Testers can use their own knowledge about the SUT to check if outputs meet the SUT specification. This test oracle is known as a human oracle (Fig. 10.3) [12]. A human oracle suffers from several disadvantages; most notably, it is likely to be error-prone and also slower than an automated check, which may restrict its application to only trivial input/output behaviour.

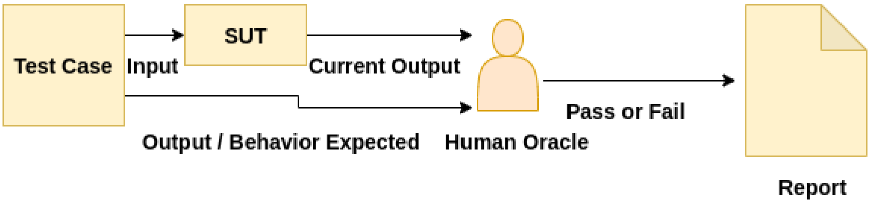

Fig. 10.3 Test Oracle Structure Using Human Oracles.

10.2.2 *Test Oracles Based on Formal Specifications*

Test oracles can be generated from formal models or specifications (Fig. 10.4) [13]. In this scenario a test oracle can be automated when a mathematical model (e.g. Finite State Machine (FSM) or Petri net) of the SUT is available for testers. Test oracles based on formal models or specifications are effective in identifying failures, but defining and maintaining formal specifications is expensive to the point that such specifications are very rare.

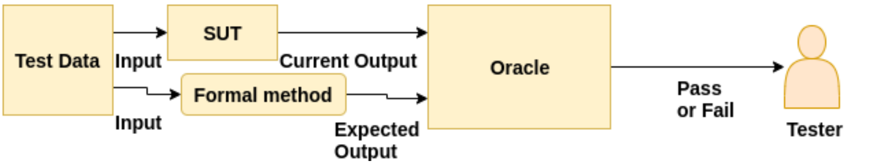

Fig. 10.4 Test Oracle Structure Using Formal Model Specification.

Figure 10.5 illustrates the structure test oracles that derive expected outputs of the SUT from test data inputs [14]. This could be possibly made by using another reference version of the SUT (e.g. an earlier version) to generate outputs from, and then the tester can build a test oracle to compare those outputs and the current outputs. In this case, testers must assume that the version used (the reference program) meets all

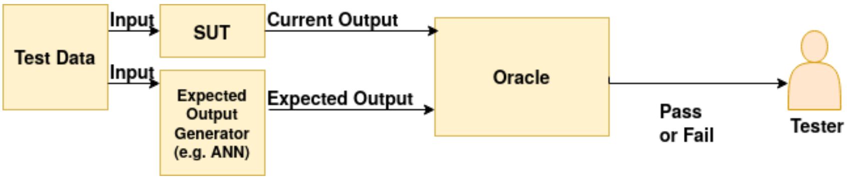

Fig. 10.5 Test Oracle Structure Using Test Data.

specifications of the SUT. This type of test oracle is widely used in the case of regression testing and mutation testing, but is not sufficient in the general case.

The oracle problem usually occurs when it is difficult to interpret test results [15]. In some cases, it is extremely difficult to predict expected behaviours of the SUT to be compared against current behaviours (this depends on the SUT) [16]. Failures can be manifested under different circumstances which makes checking the results complex or impossible to be performed [17]. Some SUTs produce outputs in very complex formats such as images, sounds or virtual environments which make the oracle problem very challenging [16].

10.3 Related Work

The automatic generation of test oracles is an important problem in software testing, but has received considerably less attention compared to other problems such as the generation of test cases. There have been three extensive reviews of topics relating to test oracles. The first by Baresi and Young [2] covered four important topics in the test oracle area: assertions, specification, state-based conformance testing and log file analysis. The second by Pezzé and Zhang [18] discussed the main techniques used to develop automated test oracles based on the available sources of information. In their survey, the source of information for test oracles was classified either as the software specification (e.g. the type of formal model specification: state-based, transition-based, history-based or algebraic) or as the program code (e.g. values from other versions, results of program analysis, machine learning models and metamorphic relations). The third by Barr *et al.* [4] classified the existing literature on test oracles into three broad categories: specified oracles, implicit oracles, and derived oracles. Specified oracles are test oracles obtained from formal specification of the system behaviour. For instance, Doong and Frankl developed the ASTOOT tool which generates test suites along with test oracles from algebraic specifications [19]. In their work, test oracles generated by the ASTOOT tool may then be used to verify the equivalence between two different execution scenarios. Specified oracles are effective in finding system failures but their success depends heavily on the availability of a formal specification of the system behaviour. However, the vast majority of systems lack an accurate, complete and up-to-date machine readable specification which limits their applicability.

Implicit oracles are generated without requiring any domain knowledge or formal specification and hence can be applied to all runnable programs. For example, in the fuzzing approach proposed by Miller *et al.* [20], the main principle is to generate random inputs and attack the system to find faults which cause the system to crash. If a crash is spotted then the fuzz tester reports the crash with the set of inputs or input sequences that caused it. The fuzzing approach is well used in the security vulnerabilities detection area such as buffer overflows and memory leaks etc. but relies on the consequences of an error being easily detectable (e.g. in the form of a system crash) so has limited general applicability.

Derived oracles are built from properties of the SUT, or several artefacts other than the specification (e.g. documentation and system execution information), or other versions of the SUT. For instance, metamorphic testing has been used to test search engines such as Google and Yahoo [21]. The BERT tool is another example of a derived oracle which can be used to identify behavioural differences between two versions of a program by using dynamic analysis [22].

Each oracle category (specified, implicit and derived) could merit an entire survey in its own right. This chapter is focused on test oracles generated using machine learning techniques (which fall into the category of derived oracles as they are typically created from system executions). Therefore, the chapter is structured according to learning strategies: supervised learning (e.g. techniques that build more on earlier versions of the SUT), semi-supervised learning (e.g. approaches that label a small number of observations and use this to seed the creation of a more complete oracle), and unsupervised learning (e.g. techniques that are based upon clustering similar results and detecting anomalies).

10.4 Test Oracles Based on Machine Learning Techniques

Chandola *et al.* defined anomaly detection as a matter of spotting patterns in data that correspond to abnormal behaviour [23]. This concept is illustrated in Fig. 10.6 where N represents regions of normal behaviour, whereas the points labelled O represent the anomalous data. The work covered in this section aimed at investigating whether software bugs generate a non-conformant pattern of behaviour that can be distinguished from the conformant or normal behaviour — in other words, in Fig. 10.6 do the groups marked N corresponded to passed tests and those marked O with failures? If this is the case then the possibility of detecting bugs automatically can be raised.

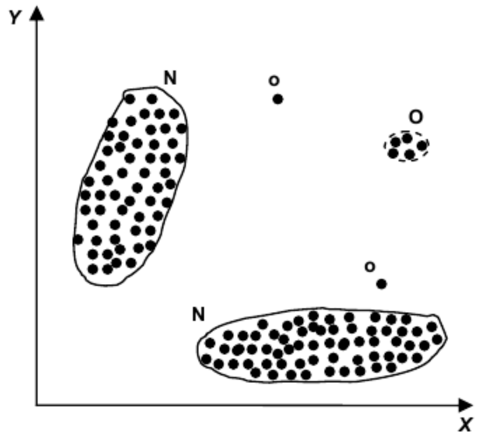

Fig. 10.6 Principle of Anomaly Detection.

The main concept behind generating automated test oracles based on machine learning techniques is to detect unexpected patterns (faulty behaviour) in a large set of observations, events or items [23]. Figure 10.7 shows the principles of using machine learning techniques to automatically cluster or classify (depending on the machine learning strategy employed) passing/failing outputs. The program under test is run on a set of inputs which will generate outputs and optional traces, and may encounter bugs in the program (the *s in the figure). The pass/fail status of the outputs is unknown and the aim is to automatically distinguish between these using machine learning. The application of anomaly detection strategies in

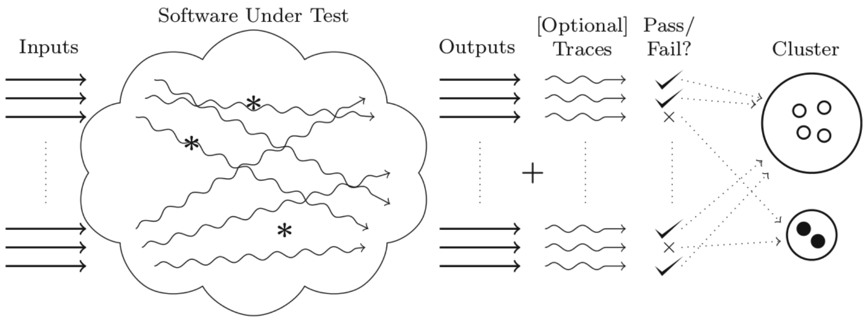

Fig. 10.7 Overview of Test Oracle Based on Machine Learning Strategy.

this context has not been extensively investigated. The current work can be divided into three main categories of learning techniques: supervised, semi-supervised and unsupervised.

10.4.1 *Test Oracles Based on Supervised Learning Techniques*

Supervised learning techniques assume the availability of a training data set which has labelled instances for normal as well as anomaly classes and is therefore the least generally applicable approach. Various Artificial Neural Network (ANN) models have been used to construct test oracles as they have the ability to simulate a software system's behaviour based on input/output pairs [14]. They can be used as continuous [24] or discrete [25] function approximators and that property can be exploited to build oracles. There are two operation phases in the development of an ANN: training and regression (or association, if the ANN is used as a classifier) [24]. Given a training set composed of input/output pairs, the ANN (in the role of a continuous function approximator) is capable of finding an approximate function of a deterministic computational process. In the role of a regression model, the trained ANN can generate the expected outputs to input data that are not part of the training set. The ANN used as a discrete function approximator can be trained with a set of input/output pairs, where the output is a category to input, and then classify other unseen inputs in one of the given categories (this scenario was illustrated earlier in Fig. 10.5 in Sec. 10.2).

For example, Aggarwal *et al.* [25], Chan *et al.* [26] and Jin *et al.* [24] tackle the use of ANN as oracles to the problem of test output classification.

Two of these papers present a case study for creating an oracle for the famous triangle problem. The inputs are three integers that represent the length sides of a triangle. The output is the classification into equilateral, isosceles, scalene or not a triangle. The ANN is given correct input/output pairs as a training set and after training phase is able to classify new inputs into the presented categories.

Vanmali and colleagues trained a multi-layer ANN on an original software application by using randomly generated test data that conformed to the specification [27]. When new versions of the original application are created and regression testing was required, the tested code was executed on the test data to yield outputs that are compared with those of the ANN. Shahamiri *et al.* [14] presented an experiment with a student registration verifier program which validates the registration, decides the maximum courses students can select and if a discount is applicable or not. A backpropagation ANN was used as an oracle and evaluated on the golden (reference) version test cases and mutated test cases.

All of previous studies used a single ANN oracle. Shahamiri *et al.* [28] proposed a multi-ANN oracle to perform input/output mapping in order to test more complicated software applications where a single ANN oracle may fail to deliver a high quality oracle. A single ANN was defined for each of the output items of the output domain; then all of ANN together made the oracle. As a result, the complexity of the software may be distributed between several ANN instead of having a single one to do all of the learning, and also separating the ANN may reduce the complexity of the training process and increase the oracle's ability to find faults. The experimental results indicate that multi-ANN oracle performed much better than single ANN oracle. However, building single ANN for every output item to create multi ANN oracle could be expensive.

Although all of previous studies demonstrate the ability of ANNs to act as a test oracle, they may not be reliable when the complexities of subject programs increase because they require larger training samples that could make the ANN learning process complicated. A small ANN error could increase the oracle miss-classification error considerably in large software applications. Moreover, most of these studies were evaluated by small subject programs having small input/output domains and the ANN was able to perform the mapping in most of these studies. It is possible that a tiny difference exists between expected output generated by the ANN based oracle and the correct program. These issues could happen because of the complexity of the application under test. Consequently, generating the most representative data sets to train the ANN could enhance the ANN performance and reduce the mis-classification error. In addition, the structure of the ANN (e.g. the number of layers and neurons) is another issue which may not be easy to determine.

More recently, Tsimpourlas *et al.* [29] use supervised learning over test execution traces. A small fraction of the execution traces were labelled with their verdict of pass or fail, and then used to train an ANN model to learn and distinguish run-time patterns for passing versus failing executions for a given program. Their experimental results showed that the classification model was highly effective in classifying passing and failing executions, achieving over 95% precision, recall and specificity while only training with an average 9% of the total traces.

ANN algorithms are not the only supervised learning models used. Wang *et al.* [30] applied support vector machine (SVM) as a supervised learning algorithm to test reactive systems. Parsa *et al.* [31] trained a support vector machine (SVM) to detect faults during the execution of subject programs. Their work was extended by using a SVM with a customised kernel function to measure the similarities between passing and failing executions, represented as sequences of program predicates [32]. Frounchi *et al.* [33] used a decision tree technique as a test oracle to verify the accuracy of an image segmentation algorithm which was able to achieve an average accuracy of approximately 90% during the evaluation phase. Brun and Ernst [34] also explored the use of SVM and decision tree to rank program properties provided by the user that are likely to indicate errors in the program.

Other learning algorithms were used by Haran *et al.* [35] to classify execution data collected from applications in the field as coming from either passing or failing program runs. They used random forests to model and predict the outcome of an execution based on the corresponding execution data. Their work was extended by proposing two different classification techniques (association trees and adaptive sampling association trees) which can build models with significantly less data than that required by random forests but maintaining the same accuracy [36]. Lo *et al.* [37] proposed a new technique to classify unknown executions. Their technique first mined a set of discriminative features capturing repetitive series of events from program execution traces. After that, feature selection was performed in order to select the best features for classification. Then, these features were used to train a classifier (SVM) to detect failures.

10.4.2 *Test Oracles Based on Semi-Supervised Learning Techniques*

Semi-supervised learning techniques are employed in situations where labelled data is scarce (e.g. instances may be difficult to come by or expensive to label). This make them very appropriate for the test oracle problem, where there may be a small set of labelled results (i.e. classified by a human as passing or failing — normal or abnormal) which may then be used to incrementally create more sophisticated classifiers.

Some of the related work in this subsection combines supervised learning with unsupervised learning, but we consider them as semi-supervised learning techniques. For instance, clustering is often performed as a preliminary step in the data mining process with the resulting clusters being used as further inputs into downstream techniques such as a neural network (it is often helpful to apply clustering analysis first to reduce the search space for the downstream algorithm). Podgurski *et al.* built a system to cluster bugs represented by a failed test that had the same cause [38]. Their approach was based on the analysis of the execution profile corresponding to reported failures of the test and was built on top of their earlier unsupervised learning system where the execution count for each function in the program was used as a feature to construct the model. Francis *et al.* proposed two new tree-based techniques for refining an initial classification of failures [39]. The first of these techniques was based on the use of dendrograms which are tree-like diagrams used to represent the results of hierarchical cluster analysis. Their dendrogram-based technique for refining failure classification was used to decide how non-homogeneous clusters should be considered for merging. The second technique for refining an initial failure classification relied on generating a classification tree to recognise failed executions. A classification tree was constructed algorithmically using a training set containing positive and negative instances of the class of interest. The experimental results indicated that both techniques were effective for grouping together failures with the same or similar causes. All techniques were aimed at bug localisation by identifying groups of failures with closely related causes among a set of reported failures based on user feedback.

Bowring and colleagues proposed an automatic classification of program behaviours using execution data aimed at reverse engineering a more abstract description of system's behaviour [40]. Their work focused on an active learning approach (rather than batch learning approach) where, for each iteration of learning, the classifier is trained incrementally on a series of labelled data elements and then applied to series of unlabelled data to predict those elements that most significantly extend the range of behaviours that can be classified. These selected elements are then labelled and added to the training set for the next round of learning. Their technique builds a classifier for software behaviour in two stages. Initially, a model of individual program executions was built as a Markov model by using the profiles of event transitions such as branches (a binary matrix was used to transform data to a suitable set of feature vectors). Each of these models thus represents one instance of the program's behaviour. The technique then used an automatic clustering algorithm to build clusters of these Markov models, which then together form a classifier tuned to predict specific behavioural characteristics of the considered program. Mao *et al.* [41] also used the Markov model approach proposed by Bowring *et al.* [40] and clustering analysis along with a new sampling strategy (priority-ranked n-per-cluster) to aid fault localisation. The methodology starts by using a Markov model and the profiles of event transitions such as branches to depict program behaviours. Based on the obtained model, the dissimilarity of two profiles is defined. After separating the failure executions and non-failure executions into different subsets, the clustering and sampling strategy were performed on the failure execution subset in order to choose the most representative sample of failures to reduce the debugging effort.

Baah *et al.* proposed a new machine learning technique that performs anomaly detection during software execution [42]. A Markov model was trained on trace predicate information and the Baum-Welch algorithm was used to find unknown parameters for the Markov model. Probes were inserted into the subject program to sample tuples in the form of <class name, method name, line number, predicate state>. Clustering of predicate states was used also in the training phase to gather predicate state information based on the line number and method number. In the line number clustering, all predicate states generated at an instrumented line number are grouped into one cluster. In method clustering, all predicate states belonging to a method are grouped into one cluster. The subject program along with the Markov model were then deployed together to detect faults as they occur and to possibly perform fault correction actions to prevent failures. The experimental results showed that the proposed technique performed well with domain faults with up to 100% accuracy in some cases. However, the technique did not perform well with computation errors with accuracy less than 50% and dropping to 0% in some cases. The authors pointed out to a few efficiency issues such as the time required to build such model especially in the presence of a large test suite, the cost of instrumenting the software to gather more information without incurring a significant overhead, and how quickly the model can track the execution of the software.

Semi-supervised learning techniques have been used as test oracles to classify passing and failing tests [43]. A learner is built by using training data set that has a small subset of labelled test data which is then used to classify the remaining data (i.e. labelling it as a passing or failing test). Different learning algorithms were explored on three systems based on dynamic execution data (firstly input/output pairs alone, and then input/output pairs combined with their corresponding execution traces). Two labelling strategies were used for the training data (labelled instances for both failing and passing tests, and just for passing tests alone). The experimental results showed that the proposed approach has an important practical implication: testers need to examine a small subset of the test results from a system and then use this information to train a learning algorithm to classify the rest. Roper [44] combined unsupervised and semi-supervised learning strategies together to construct test oracles by using the outcomes of applying unsupervised learning techniques as input to the semi-supervised learning techniques. The test classification strategy consists of two phases: Firstly, unsupervised learning (clustering) is used with the aim of creating a grouping of tests where the smallest clusters contain a greater proportion of failures. Manual checking of tests then focuses on these smallest clusters first as they are more likely to contain failing tests. Secondly, having checked a small proportion of the test outcomes, semi-supervised learning is then employed to use this information to label an initial small set of data and derive an automatic pass/fail classification for the remainder of tests. The combined effect of these is to create a far more efficient process than just checking the outcome of every test in order: clustering creates a small subset of tests in which failures are more prevalent, and using semi-supervised learning allows the tester to focus next on those outputs considered to be failures. The scenario for a test oracle based on semi-supervised learning techniques is shown in Fig. 10.8. This scenario is different compared to other scenarios in Sec. 10.2 as here the tester provides the semi-supervised learning techniques with some information to improve the oracle.

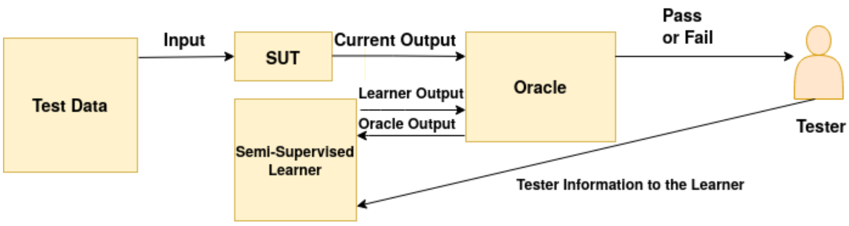

Fig. 10.8 Test Oracle Structure Using Test Data and Based on Semi-Supervised Learning Techniques.

10.4.3 *Test Oracles Based on Unsupervised Learning Techniques*

Unsupervised learning techniques do not require training data, and thus are most widely applicable. The techniques based on unsupervised learning make the implicit hypothesis that abnormal instances are relatively infrequent in the test data compared to normal instances. If this hypothesis is not true, then the techniques will suffer from high false positive rate. Dickinson, Leon and Podgurski demonstrated the advantage of automated clustering of execution profiles over random selection for finding failures by using function caller/callee feature profiles as the basis for cluster formation [45, 46]. This work is in turn based on that of Podgurski *et al.* [47], who used cluster analysis of profiles (the execution counts of conditional branches) and stratified random sampling to calculate estimates of software reliability, and found that failures were often isolated in small clusters based on unusual execution profiles.

Yoo *et al.* applied clustering to the problem of regression test optimisation [48] where test cases are clustered based on their dynamic runtime behaviour (execution traces). Their experimental results showed that the clustering approach outperformed the coverage based approach in terms of fault detection rate. Yan *et al.* [49] proposed a dynamic test cluster sampling strategy called execution spectra based sampling (ESBS). The empirical evaluation showed that the proposed sampling strategy is more effective than existing test cluster sampling strategies [45, 46]. Masri *et al.* [50] presented an empirical study of several test case filtering techniques (coverage based and distribution based techniques) by using various types of information flows (e.g. basic blocks, branches, function calls and call pairs). Their empirical study showed that coverage maximization and distribution-based filtering techniques were more effective overall than simple random sampling. In addition, distribution-based filtering techniques did not perform significantly better than coverage maximization overall.

More recently, Almaghairbe and Roper have investigated the use of clustering based anomaly detection techniques to support the construction of a test oracle by performing two different empirical studies along with varying types of dynamic execution data [51, 52]. In the first study, a range of clustering algorithms were applied to just the input-output pairs of three systems with the primary aim of exploring the feasibility of this approach. The aim of the second study was to improve the accuracy and performance of the approach by augmenting the input/output pairs with their associated execution traces. Their results demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures. Figure 10.9 shows the test oracle structure based on unsupervised learning techniques. It can be observed that the approach is built based on the SUT output only (no other information) to judge suspicious outputs. Table 10.1 summarises the difference between test oracle structures presented in this chapter.

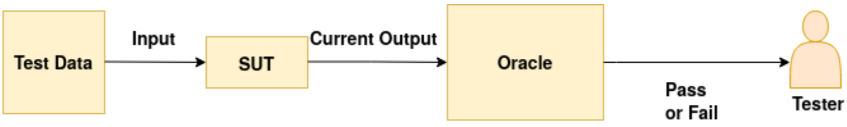

Fig. 10.9 Test Oracle Structure Using Test Data and Based on Unsupervised Learning Techniques.

10.4.4 *Summary and Findings*

Returning to the research questions in Sec. 10.1, the principal findings of this chapter may be summarised as follows:

- Question 1:
  (a) The techniques that can be used to build an automated test oracle are divided into three categories based on the learning strategies:
    i. Unsupervised machine learning techniques such as clustering methods with different sampling strategies (stratified, simple random, one-per-cluster, adaptive, n-per-cluster and failure-pursuit sampling strategies) or without any sampling strategies.

Table 10.1 The Summary of Different Concepts of Test Oracles.

Test Oracles Structures	
Test Oracles	The Different
Test oracles based on individual test cases (Fig. 10.2 and Fig. 10.3 in Sec. 10.2)	This scenario requires a tool to generate the expected output (e.g. JUnit) or tester knowledge about the SUT.
Test oracles based on formal specifications (Fig. 10.4 and Fig. 10.5 in Sec. 10.2)	This scenario requires expected output generator such as a formal method or ANN.
Test oracles based on semi-supervised learning techniques (Fig. 10.8 in Sec. 10.4)	The tester in this scenario provides semi-supervised learning techniques with some information to improve the oracle.
Test oracles based on unsupervised learning techniques (Fig. 10.9 in Sec. 10.4)	The approach in this scenario is built based on the SUT output only (no other information) to judge suspicious outputs.

ii. Semi-supervised learning techniques such as logistic regression with clustering methods and Markov model with clustering methods. Traditional semi-supervised learning techniques have been also used such as self-training and co-training methods. The results of clustering methods have also been used as input to semi-supervised/supervised learning techniques.

iii. Supervised machine learning techniques such as a multi-layered perceptron neural network, back-propagation neural network, radial basis function neural network (RBF), support vector machine (SVM) with kernel function, decision tree (DT), random forests, association trees, adaptive sampling association trees, and frequent pattern mining algorithm.

Figure 10.10 illustrates the distribution of the learning strategies of the studies discussed by counting the number of studies using each learning strategy.

(b) The chapter reported that the range of anomaly detection approaches explored were applicable for automated test oracles but different machine learning techniques with different dynamic execution data give mixed results in terms of detection accuracy. However, some techniques have been shown to be superior in term of their software fault detection accuracy. These are presented based on their learning strategy as follows:

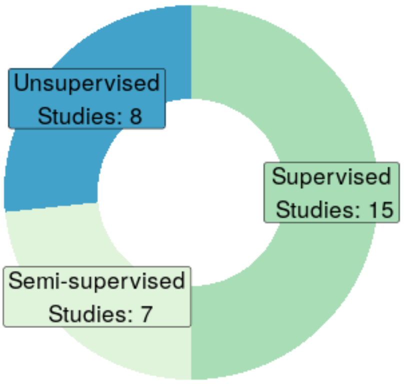

Fig. 10.10 Learning Strategies Used in the Studies.

i. Supervised machine learning techniques: Artificial neural network algorithms (ANN) showed significant accuracy in term of software fault detection for the oracle problem by using only a set of input/output pairs of SUT as feature. In addition, decision tree (DT) performed much better than support vector machine (SVM) in terms of classification accuracy by using program properties as feature. However, support vector machine (SVM) with a kernel function showed reasonable accuracy in software fault detection by using predicate state information of the program as feature. Moreover, each of random forests, association trees and adaptive sampling association trees also showed reasonable accuracy in term of software behaviour classification by using different execution data. Furthermore, frequently the pattern mining algorithm also performed well for software fault detection and localisation by using execution traces.

ii. Semi-supervised machine learning techniques: Logistic regression and clustering techniques performed well when used in combination to group failures that have the same cause together by using execution profile as the feature set. In addition, Markov model with clustering techniques performed well when used together to classify software behaviour by using both profile event transitions and predicate state information as a set of features. Furthermore, traditional semi-supervised learning techniques demonstrates their ability to compete alongside supervised learning techniques in terms of constructing an automated test oracles. In addition, the combination of clustering algorithms and semi-supervised/supervised learning techniques also showed reasonable performance.

iii. Unsupervised machine learning techniques: Clustering methods with different sampling strategies (stratified, simple random, one-per-cluster, adaptive, n-per-cluster and failure-pursuit sampling strategies) showed their ability to find failures by using different execution profiles. Simple random sampling strategy did not perform well compared to other sampling strategies. In addition, clustering algorithms with no sampling strategies proved that they can be used to build automated test oracles.

The number of individual machine learning algorithms that has been used in discussed papers is illustrated in Fig. 10.11 (fifteen Algorithms in total). From the algorithms shown in Fig. 10.9, the five most frequently individual algorithms are ANN, Hierarchical clustering with sample strategies, SVM, Markov model with clustering and DT. ANN is the most frequently used individual algorithm for test oracle construction with seven studies.

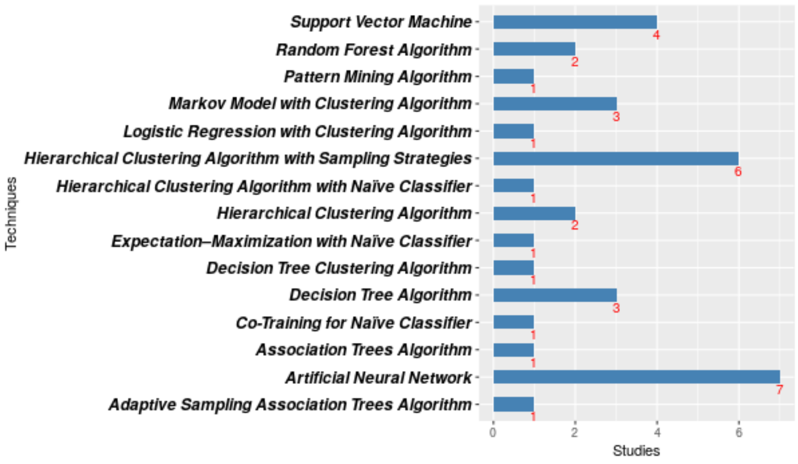

Fig. 10.11 Individual Algorithms Used for Test Oracle Construction.

(c) In terms of effectiveness, a classification strategy (supervised machine learning techniques) is reported to be better in most cases compared to a clustering strategy (unsupervised machine learning techniques) in terms of software fault detection. However, a classification strategy has disadvantages that can be summarised as follows (Chandola *et al.*, 2009): (1) multi-class classification-based techniques rely on the availability of accurate labels for various normal classes, which is often not possible; (2) classification-based techniques assign a label to each test instance, which can also become a disadvantage when a meaningful anomaly score is desired for the test instances.

As can be seen from Fig. 10.12, classification problems used far more frequently than other machine learning problems when constructing test oracles.

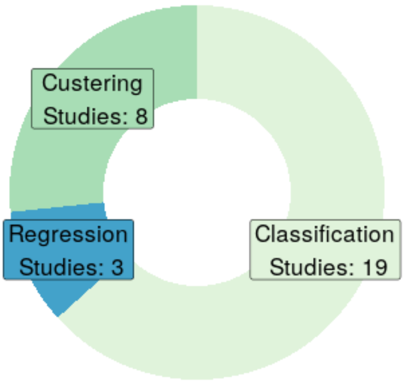

Fig. 10.12 The Frequency of Machine Learning Problem Types Used to Construct Test Oracles.

- Question 2:

  (a) The chapter reported that different types of dynamic execution data have been used as a set of features to build anomaly detection model such as the execution count of conditional branches, function caller/callee profiles, execution count for functions or methods, input/output pairs, throw counts, catch counts and execution traces such as temporal relation events, program properties, frequent path sequences, profiles of event transitions, predicate state information, and method entry/exit points.
  Figure 10.13 shows the frequency of execution data usage in discussed papers. Input/Output pairs and execution profiles are the most

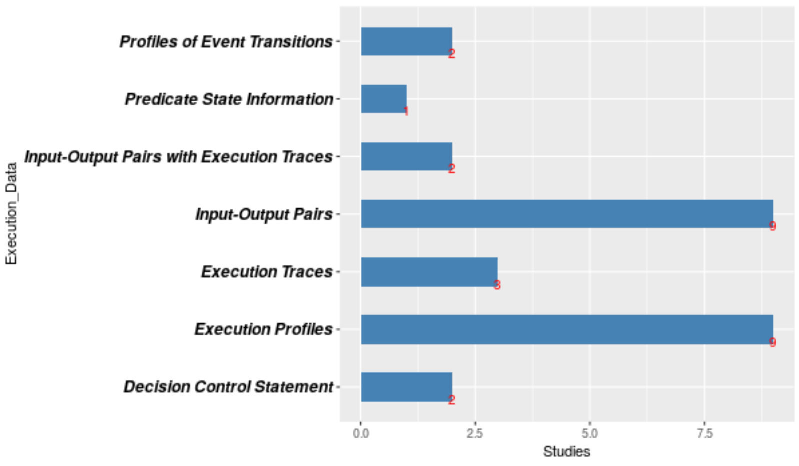

Fig. 10.13 Frequency of Use of Execution Data Types Over the Studies.

frequently used feature for building test oracles (9 studies each), followed by execution traces, combinations of input/output pairs and execution traces, decision control statements and profiles of event transition (3 and 2 studies for each feature respectively). Predicate state information is only used in 1 study.

(b) The experimental results in some studies showed that the execution count for conditional branches and execution count for functions or methods are more suitable to build an effective automated software fault detection model (using a supervised learning strategy) compared to throw count and catch count. In addition, the combination of input/output pairs and execution traces is more suitable to build an effective test oracle compared to input/output pairs alone.

(c) The chapter reported that different approaches have been used to transform dynamic execution data to suitable set of feature vectors for anomaly detection techniques to be able to make meaningful distance comparisons between execution trace profiles such as binary metric, proportional metric, SD metric, histogram metric, linear regression metric, count binary metric and proportional binary metric [45]. However, binary metric is the most commonly used approach, and normalisation has only been used in the case of using input/output data with anomaly detection to build an automated test oracle. Tokenisation and compression is also used to transform text input/output pairs and execution traces into suitable and compact feature representations.

- Question 3:
  (a) The chapter reported different types of faults that have been used in the experiments such as operator faults, domain faults, computation faults, fatal faults (executions that terminated because an uncaught exception), non-fatal faults (executions that terminated normally but produce the wrong output), omission faults (omission bugs are those where methods/functions that should have been invoked were absent), additional faults (bugs are those where methods/functions were invoked but were unnecessary and caused a failure of execution run) and ordering faults (bugs are those methods/functions that are called out of sequence).
  (b) The experimental results for some studies have reported that domain faults are more frequently detected by the Markov model compared to computation errors, and also omission and additional bugs are more frequently detected by the pattern mining algorithm compared to ordering bugs.
  (c) There is no evidence for any relationship between specific types of faults and anomaly detection approaches that have been reported in discussed papers in this chapter.

Tables 10.2–10.4 provide a summary of the available approaches, the underlying mechanisms and input/output data. In addition, Tables 10.5–10.7 highlight the main findings for all proposed approaches in the paper discussed in this chapter.

10.5 Discussion

In terms of their application in practice, the most important properties that test oracles need to demonstrate are scalability, fault detection ability, a low false positive rate and cost effectiveness. Each of those properties is explained further below:

- Scalability means the ability of any technique to handle any size of software (with corresponding increases in the volume of data). In other words, a technique has to be potentially usable at an industrial level.

Table 10.2 Summary of Selected Studies Using Supervised Learning.

Techniques Based on Supervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Jin *et al.* (2008)	Test oracle	ANN	Input/Output pairs	N/A
Aggarwal *et al.* (2004)	Test oracle	ANN	Input/Output pairs	N/A
Chan *et al.* (2006)	Test oracle	ANN	Input/Output pairs	Normalisation
Vanmali *et al.* (2002)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Shahamiri *et al.* (2010)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Shahamiri *et al.* (2012)	Test oracle	ANN	Input/Output pairs	Normalisation - Binary
Tsimpourlas *et al.* (2020)	Test oracle	ANN	Execution traces (sequences of method invocations)	Binary
Wang *et al.* (2011)	Test oracle	SVM	Execution traces (variables for temporal Relation events)	Binary
Parsa *et al.* (2009)	Faults detection	SVM	Decision control statements	Binary
Parsa *et al.* (2012)	Faults detection	SVM	Decision control statements	Binary
Frounchi *et al.* (2011)	Test oracle	DT	Input/Output pairs	Normalisation
Brun *et al.* (2004)	Finding error in program properties	SVM and DT	Execution traces (program properties)	Binary
Haran *et al.* (2005)	Classifying execution data	Random forests and DT	Statement, throw/catch and method counts	Principal component analysis
Haran *et al.* (2007)	Classifying execution data	Random forests, association trees and adaptive sampling association trees	statement, throw/catch and method counts	Principal component analysis
Lo *et al.* (2009)	Failures detection	Pattern mining algorithm	Execution traces (the frequent path sequences with timing information)	Binary

- Fault detection ability refers to the effectiveness which new (unseen) faults occurring in the running application are identified.
- False positive rate is the rate of false alarms reported by test oracles. This can be considered as the biggest issue with automated oracles. When such a rate is intolerably high, any problem reported by automated oracles will be deemed unreliable and ignored by developers.

Table 10.3 Summary of Selected Studies Using Semi-Supervised Learning.

Techniques Based on Semi-Supervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Podgurski *et al.* (2003)	Built a system to group together failures with the same or similar causes	Logistic regression and clustering technique	Execution function counts	Binary
Francis *et al.* (2004)	Built a system to classifying reported instances of software failures so that failures with the same cause are grouped together	tree-based techniques (e.g. CART algorithm) and clustering technique	Execution function/method counts	Binary
Bowring *et al.* (2004)	Automatic classification of program behaviours using execution data	Markov model and clustering technique	Profiles of event transitions such as branches	Binary
Mao *et al.* (2005)	Software faults localization	Markov model and clustering technique with priority-ranked-n-per-cluster sampling strategy	Profiles of event transitions such as branches	Binary
Baah *et al.* (2006)	Software faults detection and localization on deployment stage	Markov model with clustering technique	Predicate state information such as class name, method name, line number and predicate state	Binary
Almaghairbe *et al.* (2016)	Test oracle	Self-training and co-training techniques	Input/Output pairs	Tokenization process
Roper (2019)	Test oracle	Cluster analysis with supervised learning techniques	Input/Output pairs and execution traces (methods entry/exit points)	Tokenization process for input/output pairs and hash function for execution trace

- Cost effectiveness takes into consideration the effort and resources required to create an oracle in relation to its ability to reveal subtle semantic failures.

Generally, all those properties are complementary to each other and can affect the usability of any test oracle in practice. The ultimate goal of the software testing community is to find a test oracle that can be used to test any system, and is able to find all failures with a low false positive rate at an acceptable cost.

Test oracles based on supervised learning techniques have been widely used to build an automated test oracle. They have shown that they are

Table 10.4 Summary of Selected Studies Using Unsupervised Learning.

Techniques Based on Unsupervised Learning Strategy				
Author	Aim	Technique	Data	Transformation - Processing
Podgurski *et al.* (1999)	Estimation of software reliability	Cluster analysis with stratified sampling strategy	Execution profiles (execution counts of conditional branches)	Binary
Dickinson *et al.* (2001a)	Finding software failures	Cluster analysis with different sampling strategies (simple random, one-per-cluster, adaptive and n-per-cluster)	Execution profiles (function caller/callee)	Binary, proportional, histogram, standard deviation, linear regression, count binary and proportional count binary
Dickinson *et al.* (2001b)	Finding software failures	Cluster analysis with different sampling strategies (failure-pursuit, simple random, one-per-cluster, adaptive and n-per-cluster)	Execution profiles (function caller/callee)	Binary, proportional, histogram, standard deviation, linear regression, count binary and proportional count binary
Yoo *et al.* (2009)	Finding software failures	Cluster analysis without sampling strategy	Execution profiles (execution counts of conditional branches)	Binary
Yan *et al.* (2010)	Finding software failures	Cluster analysis with execution-spectra-based sampling strategy (ESBS)	Execution profiles (execution counts of conditional branches)	Binary
Masri *et al.* (2007)	Finding software failures	Cluster analysis via coverage based and distribution based filtering techniques	Execution profiles (basic blocks, branches, function calls and call pairs)	Numeric
Almaghairbe *et al.* (2015)	Test oracle	Cluster analysis without sampling strategy	Input/Output pairs	Tokenization process
Almaghairbe *et al.* (2017)	Test oracle	Cluster analysis without sampling strategy	Input/Output pairs and execution traces (methods entry/exit points)	Tokenization process for input/output pairs and hash function for execution trace

able to test any system with any size which make them scalable. They also tend to display a powerful ability to detect failures with a very low false positive and false negative rate (in other words, a high classification accuracy). However, their effectiveness depends on the availability of a fully labelled training data set (each instance in the training data set has to be labelled as passing or failing test execution) to construct the oracle. Labelling each instance in the training data set can be an expensive process and typically relies on using a reference version of the software (which is difficult to obtain in practice), making them prohibitively expensive and not cost-effective.

Table 10.5 Summary of Findings for Selected Studies Using Supervised Learning.

Techniques Based on Supervised Learning strategy	
Author	Findings
Jin *et al.* (2008)	The average accuracy for the technique is approximately 60% over several data sets.
Aggarwal *et al.* (2004)	The mean miss-classification error for the technique is 15.9% with standard deviation of 2.312. The mean non-prediction error for the technique is only 2.949% with standard deviation of 1.252. The total mean relative error is found to be 19.02% with standard deviation of 2.224.
Chan *et al.* (2006)	The average accuracy for the technique is 78.14%.
Vanmali *et al.* (2002)	The minimum classification error rate for the technique for the binary output is 8.31% and for the continuous output is 20.79%.
Shahamiri *et al.* (2010)	The average accuracy for the technique is 95.37%.
Shahamiri *et al.* (2012)	The average accuracy for the technique in the first case study is 95.7% and for the second case study is 98.83%.
Tsimpourlas *et al.* (2020)	The classification model for each of the subject programs is highly effective in classifying passing and failing executions, achieving over 95% precision, recall and specificity while only training with an average 9% of the total traces.
Wang *et al.* (2011)	The average accuracy for the technique is 95.46%.
Parsa *et al.* (2009)	The average precision for the technique is 85%.
Parsa *et al.* (2012)	The average precision for technique is 65%.
Frounchi *et al.* (2011)	The average accuracy for the technique is 95%.
Brun *et al.* (2004)	The experimental results showed for C programs, 45% of the top 80 properties are fault-revealing. For Java programs, 59% of the top 80 properties are fault-revealing. The DT technique performed much better than SVM in term of classification. The SVM technique performed much better than DT technique in term of ranking properties.
Haran *et al.* (2005)	Statement counts and method counts succeeded in building classification model with higher accuracy compared to the classification model used throw counts and catch counts.
Haran *et al.* (2007)	Three techniques performed well with overall miss-classification rates typically below 2 percent.
Lo *et al.* (2009)	The average accuracy for proposed algorithm was 93.77% and 24.67% over the base classifier. The technique worked well with omission and additional bugs but poorly with ordering bugs.

Table 10.6 Summary of Findings for Selected Studies Using Semi-Supervised Learning.

Techniques Based on Semi-Supervised Learning Strategy	
Author	Findings
Podgurski *et al.* (2003)	These results indicate that the strategy can be effective and is able to group together failures with the same or similar causes.
Francis *et al.* (2004)	Experimental results indicate that the proposed techniques are effective for grouping together failures with the same or similar causes.
Bowring *et al.* (2004)	An active learning approach had very high classification accuracy and performed better than batch learning approach.
Mao *et al.* (2005)	The results show that the clustering and sampling techniques based on revised Markov model is more effective to find faults than Podgurski's method (one-per-cluster sampling method).
Baah *et al.* (2006)	The technique performed well with domain faults and performed poorly with computation error.
Almaghairbe *et al.* (2016)	The results show that in many cases labelling just a small proportion of the test cases as low as 10% is sufficient to build a classifier that is able to correctly categorise the large majority of the remaining test cases.
Roper (2019)	The accuracy of the technique is 86% with just only 31 test cases labelled.

Test oracles based on semi-supervised learning techniques are less expensive in comparison to those based on supervised learning techniques as they require a smaller set of labelled training data (as opposed to the large data set required by supervised techniques or the fault-free version employed by invariant detectors). However, test oracles based on semi-supervised learning techniques have a lower accuracy in comparison to those based on supervised learning techniques (they have a slightly higher false positive rate, and also slightly lower fault detection ability) which is to be expected as the training of the algorithms use far less labelled data. Semi-supervised approaches are a classic demonstration of the cost benefit trade-off: a larger set of labelled data is likely to yield a more accurate classifier, and while these techniques are significantly more cost-effective (and practicable) than supervised approaches, there is still work to be done in establishing the ideal ratio of labelled to unlabelled data.

Test oracles based on unsupervised learning techniques do not require the availability of labelled data or a fault free version of the SUT to construct test oracles which make them more scalable in comparison to test oracles based on supervised/semi-supervised learning techniques and test oracles based on invariant detection in terms of the provision of labelled data (other scalability issues may arise in the application of the algorithms

Table 10.7 Summary of Findings for Selected Studies Using Unsupervised Learning.

Techniques Based on Unsupervised Learning Strategy	
Author	Findings
Podgurski *et al.* (1999)	Experimental results suggest that the approach is computationally feasible, can isolate failures, and can significantly reduce the cost of estimating software reliability. Stratified sampling strategy performed much better than simple random sampling strategy.
Dickinson *et al.* (2001a)	The experimental data shows that the percentage of failures found in the smallest clusters is significantly higher than 50%. One-per-cluster, adaptive and n-per-cluster sampling strategies performed much better than simple random sampling strategy. Also, binary, standard deviation, histogram, proportional and proportional binary metrics with the sampling strategies performed much better than linear regression and binary metrics.
Dickinson *et al.* (2001b)	The experimental data shows that the percentage of failures found in the smallest clusters is significantly higher than 50%. Failure-pursuit, one-per-cluster, adaptive and n-per-cluster sampling strategies performed much better than simple random sampling strategy. Also, binary, standard deviation, histogram, proportional and proportional binary metrics with the sampling strategies performed much better than linear regression and binary metrics.
Yoo *et al.* (2009)	The empirical studies show that the proposed approach (hybrid ICP algorithm) can outperform the traditional coverage-based prioritisation for some programs.
Yan *et al.* (2010)	The experimental results show that clustering algorithm with ESBS sampling strategy is better in failures detection than existing sampling strategies in most case.
Masri *et al.* (2007)	Both coverage maximization and distribution-based filtering techniques were more effective overall than simple random sampling, although the latter performed well in one case in which failures comprised a relatively large proportion of the test suite. In addition, distribution based filtering techniques did not perform significantly better than coverage maximization overall.
Almaghairbe *et al.* (2015)	The findings reveal that failing outputs do indeed tend to congregate in small clusters, suggesting that the approach is feasible and has the potential to reduce by an order of magnitude the numbers of outputs that would need to be manually examined following a test run.
Almaghairbe *et al.* (2017)	The experimental results gave an evidence which support the clustering hypothesis behind their work where in several cases small (less than average sized) clusters contained more than 60% of failures (and often a substantially higher proportion). As well as having a higher failure density they also contained a spread of failures in the cases where there were multiple faults in the programs. The results also demonstrate important practical consequences: the task of checking test outputs may potentially be reduced significantly to examining a relatively small proportion of the data to discover a large proportion of the failures.

but these are likely to be equally applicable to all approaches). In addition, they are less expensive to obtain in comparison to test oracles based on supervised/semi-supervised learning techniques (again as no data labelling is necessary), but can be less accurate for the same reason.

Most recently, Almaghairbe *et al.* [53, 54] empirically evaluated test oracles based on semi-supervised and unsupervised learning techniques by comparing them with existing techniques from the mining specification domain (the data invariant detector Daikon [55]). The experimental results showed that semi-supervised learning techniques have a higher fault detection ability (82% on average) and lower false positive rate (11% on average compared to 16% for Daikon and for unsupervised learning). In addition, unsupervised learning techniques have slightly higher fault detection ability compared to Daikon (66% compared to 64%), but a similar false positive rate (16% on average).

In general, as such approaches are developed, more work is needed on the measurement of oracles and their properties, and also it has been suggested that it is important for the software metrics community to consider the concept of "oracle metrics" [4].

10.6 Further Research Direction

Despite the achievements identified in this chapter, there are a number of barriers that need to be overcome for the work to become practically usable which fall into the categories of scalability and accuracy. These are explained further below:

10.6.1 *Improving the Accuracy*

Accuracy in this context means the ability of the proposed approaches to identify failing and passing test results as correctly as possible (high true positive rate and low false positive/false negative rates). The accuracy of semi-supervised and unsupervised learning techniques may be improved by augmenting the data sets (input/output pairs and execution traces) with more relevant information from the program execution (e.g. state information, execution time and code coverage etc.) to build more effective/accurate test oracles, but this strategy still needs to be evaluated.

Adding more execution data to the data sets can help to reduce the size of labelled data needed to train the learning algorithms in semi-supervised learning. This also relates to scalability but to make the approach practical the size of labelled data needs to be as low as possible which means improving the accuracy as well. The other point related to the size of labelled data and then accuracy is that the predictions of semi-supervised learning approaches should come with an estimate of confidence.

The accuracy of unsupervised learning approaches can be improved by selecting the most appropriate similarity measures for clustering algorithms. In addition, the number of specified clusters for clustering algorithms is important and the accuracy can be improved by specifying the optimal number of cluster counts as this can have a significant impact on the way that passing and failing tests become grouped into different clusters.

The final way to improve accuracy is to investigate alternative algorithms. A wide range have already been employed as has been seen in this study but there are many machine learning algorithms which have yet to be explored, along with meta-level approaches such as ensemble methods which combine learners to build more effective and accurate algorithms.

10.6.2 *Improving the Scalability*

Many of the approaches reviewed work with very complex feature sets: test inputs and outputs, execution traces, function call profiles, predicate state information, event transition profiles etc. These are often extensive sets of information which are not easily handled by machine learning algorithms and consequently need to be transformed and encoded in a form that allows them to be compared sensibly and used meaningfully as feature sets. For instance, the input/output pairs for tested systems in [43, 51, 52] were of string/text type and it turned out that a tokenization procedure worked reasonably well with the proposed approach, but this may not be generally applicable for all input and output types. More general strategies need to be identified to make the approaches scalable and widely applicable.

An interesting approach to addressing the scalability (and cost-effectiveness) issue which should be explored in the future is to investigate the feasibility of using the cheap results from clustering in which there is the greatest confidence to generate the labelled set required by the semi-supervised techniques, and thereby reduce the cost of the semi-supervised learning. There has already been some initial work in this area [44]

In summary, the fundamental principle to the successful automated test oracles is the capability to build oracles that demonstrate a substantially lower false positive rate and higher fault detection capability, as compared to the available, state of the art tools. Therefore, future research is also devoted to further development and empirical investigation of the effectiveness of several automated test oracles, to evaluate the features offered by different alternative oracles.

10.7 Conclusion

The importance of testing activity is widely known. However, the difficulty in deciding whether a result is acceptable or not (also known as the oracle problem) hampers such activity.

The comparison between expected output and obtained output is often determined manually by the tester, which is usually slow, error-prone, and very expensive. Several approaches have been proposed in order to address the oracle problem and automate the process.

The main contribution of this chapter is to present a comprehensive overview on test oracles based on machine learning techniques, and identify the strengths and limitations of the approach. We also provided a discussion about the required properties of automated and semi-automated test oracle techniques (based on machine learning techniques) for them to be practically usable such as scalability, accuracy and cost effectiveness. Those properties are used as criteria to evaluate the existing approaches. To conclude, the chapter sets out a road map for future work and a guideline to software testing researchers who seek to address this challenging and important problem.

References

[1] G. Fraser and A. Arcuri, Evosuite: Automatic test suite generation for object-oriented software, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. ACM, New York, NY, USA, ISBN 978-1-4503-0443-6, pp. 416–419 (2011), ISBN 978-1-4503-0443-6, doi:10.1145/2025113.2025179, `http://doi.acm.org/10.1145/2025113.2025179`.

[2] L. Baresi and M. Young, Test oracles, Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A. (2001), `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[3] C. Pacheco and M. D. Ernst, Randoop: Feedback-directed random testing for java, in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07. ACM, New York, NY, USA, ISBN 978-1-59593-865-7, pp. 815–816 (2007), ISBN 978-1-59593-865-7, doi:10.1145/1297846.1297902, `http://doi.acm.org/10.1145/1297846.1297902`.

[4] E. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, The oracle problem in software testing: A survey, *Software Engineering, IEEE Transactions on* **41**, 5, pp. 507–525 (2015), doi:10.1109/TSE.2014.2372785.

[5] D. Tu, R. Chen, Z. Du and Y. Liu, A method of log file analysis for test oracle, in *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on*, pp. 351–354 (2009), doi:10.1109/EmbeddedCom-ScalCom.2009.69.

[6] A. Memon, I. Banerjee and A. Nagarajan, What test oracle should I use for effective gui testing? in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 164–173 (2003).