Technical Report OBU-ECM-AFM-2018-01

Software Testing as A Problem of Machine Learning -- A Survey and Critical Review

Hong Zhu

Applied Formal Method Research Group
School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK

Tel.: ++44 01865 484580, Email: hzhu@brookes.ac.uk

12 December 2018

Submitted to *The Journal of Software Testing, verification and Reliability*

Abstract

This paper reviews the research on employing machine learning (ML) to solve software testing problems. Existing works can be classified into four categories according to the type of testing problems addressed: (a) adequacy problem, i.e. how to measure test adequacy, (b) oracle problem, i.e. how to determine the correctness of software's outputs on test cases, (c) design problems, i.e. how to generate, prioritise and optimise test cases, and (d) testability problem, i.e. how to measure software's testability. For each of these problems, various approaches proposed and studied in the literature are identified, summarised and critically analysed. Three observations are made on the current state of art and future research directions are identified: (1) *No easy money*. Applying ML techniques to testing is labour intensive and expensive. It is not economically viable as a practical technique, yet. (2) *No free lunch*. Wolpert's *No-Free-Lunch* theorem is observed; that is, there is no single ML algorithm that consistently performed better than other ML techniques. (3) *No silver bullet*. Existing ML techniques cannot deal with the complexity confronted software engineers in current practices. Although ML in its current state is still far from a practically applicable testing technique, it can potentially bring significant and profound impacts on both research and practice. In particular, it is worth further exploring a novel theoretical model of software testing in parallel with the PAC learning theory by treating testing as inductive inference. A break through in this direction could foster a paradigm shift in testing and reliability assessment.

Keywords: Software Testing, Machine learning, Learnability, Neural networks, Inductive inference, Clustering, Data mining, Test adequacy, Test oracle, Test case generation, Test case prioritization, Test case reduction, Testability.

Table of Contents

Software Testing as A Problem of Machine Learning -- A Survey and Critical Review

Hong Zhu
School of Engineering, Computing and Mathematics
Oxford Brookes University, Oxford OX33 1HX, UK
Tel.: ++44 01865 484580, Email: hzhu@brookes.ac.uk

Abstract

This paper reviews the research on employing machine learning (ML) to solve software testing problems. Existing works can be classified into four categories according to the type of testing problems addressed: (a) adequacy problem, i.e. how to measure test adequacy, (b) oracle problem, i.e. how to determine the correctness of software's outputs on test cases, (c) design problems, i.e. how to generate, prioritise and optimise test cases, and (d) testability problem, i.e. how to measure software's testability. For each of these problems, various approaches proposed and studied in the literature are identified, summarised and critically analysed. Three observations are made on the current state of art and future research directions are identified: (1) *No easy money*. Applying ML techniques to testing is labour intensive and expensive. It is not economically viable as a practical technique, yet. (2) *No free lunch*. Wolpert's *No-Free-Lunch* theorem is observed; that is, there is no single ML algorithm that consistently performed better than other ML techniques. (3) *No silver bullet*. Existing ML techniques cannot deal with the complexity confronted software engineers in current practices. Although ML in its current state is still far from a practically applicable testing technique, it can potentially bring significant and profound impacts on both research and practice. In particular, it is worth further exploring a novel theoretical model of software testing in parallel with the PAC learning theory by treating testing as inductive inference. A break through in this direction could foster a paradigm shift in testing and reliability assessment.

Keywords: Software Testing, Machine learning, Learnability, Neural networks, Inductive inference, Clustering, Data mining, Test adequacy, Test oracle, Test case generation, Test case prioritization, Test case reduction, Testability.

1. Introduction

Can machine learning (ML) solve problems of software testing? This paper reviews the current state of research in an attempt to identify the potential future directions.

By machine learning, we meant the group of artificial intelligence techniques and their underlying theories that produce a general rule from a finite subset of the instances of the rule. A typical example of such techniques is *inductive inference*, which infers a mathematical function or relation from a finite set of carefully selected instances of the function/relation. For example, learning a language's grammar from a finite set of sentences obtained by a certain way of sampling is a typical inductive inference problem. Another example of ML techniques is *data mining*, which discovers unknown patterns/rules in a set of observed data on certain phenomena. A particular type of ML techniques that is closely related to the theme of this paper is *program synthesis*, which aims at producing a program in a given programming language from a finite set of instances of input/output pairs. ML has been an active research area of artificial intelligence in the past five decades. In recent years, it has been wide applied partly because various ML tools and systems have become widely available for solving practical problems, and partly because a large volume of data has become available for many application domains.

Software testing is indispensable to all software development. However, it is labour intensive, costly, and error-prone. It is one of the most studied topics in software engineering. Attempts to solve problems in software testing by employing ML techniques can be back dated to 1980s, for example, to solve test adequacy problems [58]. Since then, there are continuous efforts reported in the literature. Among the most notable work reported in the literature are those experiments on the uses of ML techniques to develop test oracles [5], to measure test adequacy[22] [23], to generate test cases [48], to reduce test costs [20], and to understand the notion of software testability [18], etc. In general, software testing is an inductive inference

process in the course of which the tester attempts to deduce general properties of a software system (such as correctness and reliability) by observing the behaviours of the system on a finite number of test cases [68]. The theories of ML have also being employed to answer fundamental theoretical questions about software testing, such as "can testing guarantee software correctness?" [66]. In recent years, the application of ML techniques to software testing has become more active, but the progress is slow. It is desirable to review the current state of research and draw a road map for future development.

The scope of this paper will cover all aspects of the application of ML to software testing, from theoretical studies to empirical experiments. We will identify the existing approaches to various types of software testing problems and discuss their potential benefits and limitations. Four types of software testing problems are surveyed: adequacy problems, oracle problems, test design problems, and testability problems.

The paper is organized as follows. Section 2 is a brief introduction to the theories and techniques of ML in the context of software testing. Section 3 reviews the work on application of ML techniques and theories to test adequacy problems. Section 4 is devoted to the test oracle problems. Section 5 is concerned with the application of ML to the test design problems. Section 6 is about testability. Section 7 concludes the paper with a summary of the current state of art and a discussion of future research directions.

2. Introduction of ML Theories and Techniques

The research on ML back dates to 1960s and many pioneering work emerged in 1980s, which laid the foundations for the current development of inductive inference, data mining and data analytics techniques. This section is a brief introduction to the theories and techniques of ML in the context of research on their applications in software testing. It serves as the bases for the description and discussion of the work on software testing by defining the terminologies and the providing the basic theorems. Being an active research area for more than 50 years, a thorough survey of the topic is beyond the scope of this paper. The readers are referred to the literature of ML, e.g. [34], [36] and [51], for more systematic and in depth coverage of the topic. Readers who are familiar with ML theories and techniques can skip this section.

2.1 Theories of Machine Learning

This subsection is a very brief introduction to two theoretical frameworks of ML. We will start with definitions of the basic concepts of ML theories, then give brief introductions to two widely used theories of ML: (a) the theory of learnability in the *identification in the limit*, (b) the theory of learnability in *PAC learning*.

2.1.1 Basic Concepts of ML

There are a number of different abstract models of ML, which are also called *inductive inference protocols* or *ML schemes* in the literature. Although these models differ from each other significantly, they share some common basic concepts, which are introduced below.

- *Object Space.*

The object space is the set of objects to learn, which are often called the *target rules to learn*, or simply the *targets*. Typical examples of object spaces studied in ML literature are:

a) the set of all one-variable polynomial functions,

b) the set of all Boolean functions,

c) the set of threshold functions on the unit interval of real numbers,

d) the set of close intervals $[a, b]$ of real numbers,

e) the set of context-free languages on a given alphabet,

f) the set of regular languages on a given alphabet, etc.

Typically, a rule r in an object space is assumed to be a function from a domain D to a codomain C. For example, a regular language l can be defined as a function from strings of the alphabet to $\{0,1\}$ such that $l(s) = 1$ if and only if string s is a valid sentence of the language l. Often, the codomain is assumed to be

the set $\{0, 1\}$. For example, a function $f: \mathbb{N} \to \mathbb{N}$ mapping from natural numbers to nature numbers can be equivalently represented as a function $\phi_f: \mathbb{N}\times\mathbb{N} \to \{0,1\}$ such that $\phi_f(x, y) = 1$ if and only if $y = f(x)$.

- *Hypothesis Space.*

The result of a learning process is a description of the object to learn. Such a description represents the rule to learn in certain format. For example, a regular language can be represented in the form of a regular expression. An threshold function $f(x) = x > \alpha$ can be simply represented by a real number α. The set H of all possible descriptions of the objects in the object space R is called the *hypothesis space*. In many cases, a rule r in R may have many syntactically different but semantically equivalent representations in the hypothesis space. A hypothesis h may describe a rule r correctly according to a correctness criterion; see discussion below. If such a correct hypothesis is not always available in H for all rules in R, the learning problem is called *agnostic*.

- *Instances*.

An inductive inference/learning process takes a sequence (sometimes, a set) of the instances of a rule r in R as the input. The form of instances varies in different inductive inference models. Typically, an instance can be a pair $< x, y >$, where x is in the domain D of the rule r and y is the corresponding value of r in the codomain C, i.e. $y = r(x)$. Such instances are called *positive examples,* or simply *examples* for short. Another form of instances is *counterexamples*, which is a pair $< x, y >$, where x is the input in the domain D, but y in the codomain C is *not* the corresponding value of r, i.e. $y \neq r(x)$. Examples and counterexamples are often used in the so-called *supervised learning*.

Some learning algorithms require the input of instances to be in certain order. For example, to learn a total function f on natural numbers, one may require that the instances are in the form of $< 0, f(0) >, < 1, f(1) >, \dots, < n, f(n) >, \dots$. Other may be able to take any sequence of examples of the function as input in any order. Another way is that the learning algorithm queries about the value of the rule on a specific point of the domain.

In clustering and classification type of ML processes, the input often contains only a set of values in the domain sampled according to certain probability distribution without the corresponding values of the rule. These values represent a set of entities to be classified. In such cases, assumptions on the rule space are made, for example, the similarity between the entities can be measured by a given distance function on the data.

- *Inference Device.*

An inductive inference device M is a computational device, such as a Turing machine, an algorithm, or a neural network, etc. It takes as input a sequence of the instances of a target rule r in the object space R, and produces a description h of the rule r in the hypothesis space H as the result of learning.

- *Correctness criterion*.

Usually, the hypothesis h produced by an inductive inference device M is required to be correct on the input instances, i.e. h is consistent with the rule r on the input instances, or within a tolerable margin of error. Whether the produced hypothesis is acceptable is judged according to a *correctness criterion*.

One of the most often used correctness criteria is *logical equivalence*, written $r = h$, i.e. for all data x in D, $r(x) = h(x)$. Sometimes a finite number of anomalies are allowed. Let n be a natural number, $r =_n h$ is defined as follows:

$$r =_n h \iff |\{x \in D | r(x) \neq h(x)\}| \leq n.$$

And, $r =_* h$ is defined as follows.

$$r =_* h \iff \{x \in D | r(x) \neq h(x)\} \ is \ finite.$$

The most commonly used correctness criterion in statistical learning is defined in the form of the probability of correctness. Let P be a probability distribution on D. A hypothesis h is *probably correct* w.r.t. rule r with probability better than ε, written as $r =_{(P,\varepsilon)} h$, if

$$Pr(\{x \in D | r(x) \neq h(x)\}) \leq \varepsilon.$$

Another important correctness criterion is concerned with the loss or risks of failure. Let $l: D \times C \times C \rightarrow \mathbb{R}^+$ be a loss function. A risk function $Risk_l(r, h)$ can be defined as the expected loss for a hypothesis h with respect to a rule r as follows.

$$Risk_l(r, h) = Ex_{Pr}\left(\left\{l\left(x, h(x), r(x)\right) \middle| x \in \widehat{D}(r, h)\right\}\right),$$

where $\widehat{D}(r, h) = \{x \in D | \, r(x) \neq h(x)\}$.

- *Convergence criterion.*

In general, inductive inference is a process that takes more and more instances of the rule to learn and produces a sequence of hypothesis. A convergence criterion determines when an inductive inference process reaches a conclusion, and thus can stop. It determines how close the hypothesis produced is to the rule to learn. The next subsection reviews various convergence criteria together with the abstract models of ML and the corresponding notion of learnability.

2.1.2 Identification in The Limit

One of the most well-known and earliest abstract models of ML is *identification in the limit*. It has its origin in the work of Gold in 1967 on learning languages [27]. It views inductive inference as an infinite process in which an inductive inference device *M* is run repeatedly on more and more of examples of a rule r, say $x_1, x_2, \ldots, x_n, \ldots$, and it outputs an infinite sequence of *hypothesis* $h_1, h_2, \ldots, h_n, \ldots$. If the inductive inference device makes the same hypothesis after some finite number m of changes, the process is defined as convergent after m changes. Identification in the limit can be formally defined as follows.

Definition 2.1 (*Identification in the Limit*)

Suppose that M is an inductive inference device, R is a set of rules, $r \in R$ is a rule to learn, and $\vec{x} = x_1, x_2, \ldots, x_n, \ldots$ be an infinite sequence of instances of rule r. Let $h_n = M(\{x_1, \ldots, x_n\})$. If there exists some natural number m such that, for all $i > 0$, $h_m = h_{m+i}$, then we say that M converges to h_m, written as $M(\vec{x}) \downarrow h_m$. If $h_m \equiv r$, we say that *M behaviourally* identifies r in the limit. If $h_m = r$, we say that *M explanatorily* identifies r in the limit. □

Definition 2.2 (*Behavioural or explanatory identifiability*)

If there is an inductive inference device M such that for every rule r in R, r is behaviourally (explanatorily) identifiable in the limit by M, we say that R is *behaviourally* (*explanatorily*) *identifiable*. □

Let *BC* and *EX* denote the collections of rule sets R that are behaviourally and explanatorily identifiable in the limit, respectively. Moreover, let n be any given natural number. If replacing the equals to $=$ relation in the definition of *EX* with relation $=_n$ and $=_*$, the collections of identifiable rule sets are denoted by EX^n for each natural number n and EX^*, respectively. Similarly, we define BC^n and BC^*. Case and Smith (1978, 1983) proved the following theorem about the learnability of identification in the limit [14].

Theorem 2.1 (Case and Smith 1978, 1983)

$$EX = EX^0 \subset EX^1 \subset \cdots \subset EX^n \subset \cdots \subset \bigcup_{n \in N} EX^n \subset EX^*$$

$$\subset BC = BC^0 \subset BC^1 \subset \cdots \subset BC^n \subset \cdots \subset \bigcup_{n \in N} BC^n \subset BC^*$$

□

An example of learnable sets of rules in EX is the set $Poly$ of one-variable polynomial functions, i.e. $Poly \in EX$. An example of an unlearnable set of rules for BC is the set Tol of total computable functions, i.e. $Tol \notin BC$, but Tol is learnable with respect to BC^*, i.e. $Tol \in BC^*$.

Identification in the limit was originally proposed by Gold for the study of the computational learning of language grammars [27]. The learnability of various classes of languages depends on the learning model which consists of two factors: (a) how the instances of the language are presented to the learner, and (b) the

representation of learning outcomes. Gold used two representations of the learning outcomes:

- Turing machines as the generators of the languages;
- Turing machines as the decision procedures for the languages.

He used the following three presentations of the input.

- *Text presentation*: The learner is presented with a text, which is a sequence $x_1, x_2, \ldots, x_n, \ldots$, of valid sentences in the language. That is, the instances only contain positive examples. In general, the text can be any arbitrary function from N to the sentences. If the sequence is a recursive function (or primitive recursive function) from N to the set of sentences in the language, the text is called *recursive text* (or *primitive recursive text*).
- *Complete presentation*: The learner is presented with a sequence of instances of the language; some are positive and some are negative.
- *Request presentation*: The learner makes a sequence of queries about whether a text is a valid sentence of the language, and presented with answers of "*yes*" or "*no*", where "*yes*" for the text being a positive example, or "*no*" for a counterexample.

Although there are six combinations of these two factors, Gold proved that there are only three learning models that have different learning power. These learning models are:

- *Anomalous Text* learning model, in which the learner is presented with primitive recursive text of the language as input and produces hypothesis in the form of a Turing machine that enumerate the language.
- *Informant* model, which uses the complete text as the input to the learner in the form of requests on whether a specific sentence is an instance of the language and get replies of yes or no. It generates an output in the form of a generator.
- *Text* model, which uses primitive recursive text as the input, but produces a Turing machine as the decision procedure for the language.

The main results of Gold's work [27] are shown in Figure 1. On the right-hand side, various language classes are listed in the order of set inclusion, i.e. a class of language is a superclass of all classes below it. On the left-hand side are three different learning models showing the classes of languages that are learnable. *Anomalous Text* learning model is the most powerful one among these three models. *Informant* model is less power than Anomalous Text model, while Text is the weakest one, which can only learn finite languages by identification in the limit.

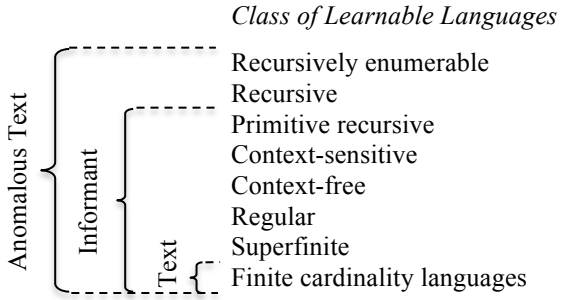


Figure 1. Learnable Language Classes by Identification in The Limit [27]

Language identification in the limit is relevant to the application of ML theories and techniques to software testing, because finite state machines, automata, and flow graph models are widely used in software testing. Whether a type of software model is learnable can be derived from the learnability of their corresponding language classes. Useful results on the characteristics of learnability can be found in Angluin's work [8].

Identification in the limit is useful to answer theoretical questions about ML, but it has limitations in practical applications. As Gold pointed out [27], even if a language belongs to a learnable class, one would not necessarily be able to know when the hypothesis produced by the learning device is correct. The inductive inference process must go on forever because there is always the possibility that the input

instance will appear that forces the learning device to change the output hypothesis. If it is required to know when the output hypothesis is correct, the inductive inference model is called *finite identification*, and then, none of the language classes in the above table are learnable. *Fixed time identification* is an abstract model that is even weaker than finite identification. It requires the language can be learned after a fixed number of steps specified a priori and independent of the specific language to learn. Gold justified the use of identification in the limit as a right model to study the learnability of language classes by arguing that "*a person does not know when he is speaking a language correctly; there is always the possibility that he will find that his grammar contains error. But, we can guarantee that a child will eventually learn a language, even if it will not know when it is correct*". Applying the same argument to software testing, we have that "*a tester does not know when the program passed his test is correct; there is always the possibility that he will find that his program contains error. But, we can guarantee that a tester will eventually get a correct program, even if it will not know when it is correct.*"

Based on identification in the limit, Gold also defined the notion of *effective* identification in the limit if there is an algorithm that computes the inductive inference device. Otherwise, it is ineffective.

2.1.3 Probably Approximately Correct (PAC) Learning

PAC learning is an abstract model of ML proposed by Valiant in 1984 [54]. In a PAC learning process, examples of the unknown target rule to learn are generated at random according to a given probability distribution, but unknown to the induction device, on the example space. The inductive inference device is given two real number parameters: δ for the confidence in the inference result and ε for the reliability of the result, both are real numbers greater than 0 and less than 1. A natural number n is calculated from ε and δ. The inductive inference device asks the environment to input n examples of the target rule r generated at random. The inference machine then produces a hypothesis h of the rule. The inference is said to be successful at learning the rule r if it outputs a hypothesis h such that, with probability at least $1-\delta$, the likelihood that h is incompatible with the next randomly generated example is at most ε. Perhaps surprisingly, the number n of input examples is not dependent of the probability distribution. Formally,

Definition 2.3 (PAC Learnability)

A class R of rules with domain D and codomain C is *PAC learnable* if there exists a function $m_R: (0,1)^2 \rightarrow N$ and an inductive inference device M with the following property: for every $\epsilon, \delta \in (0,1)$, for every probability distribution Pr over D, for every r in R, and every set t of $n = m_R(\varepsilon, \delta)$ examples of rule r, if the elements in t are drawn at random independently and identically distributed (i.i.d) according to Pr, the likelihood that the following inequality holds is greater than or equal to δ when x is drawn at random i.i.d. over D with distribution Pr.

$$Pr(\{x \in D | r(x) \neq h(x)\}) \leq \varepsilon.$$

where $h = M(t)$. The smallest such integer-valued function m_R for all inductive inference devices that R is learnable is called the *sample complexity* of R. □

The PAC learnability for a set R of rules is characterized by the *VC dimension* of the set R. Let R be a class of functions from D to $\{0,1\}$, and let $A = \{a_1, a, \dots, a_m\} \subset D$ be a finite subset of D. The restriction of R on A is defined as the set of functions from A to $\{0,1\}$ that are derived from R. That is,

$$R_A = \{< r(a_1), r(a_2), \dots, r(a_m) > | r \in R\},$$

where each function from A to $\{0,1\}$ is represented as a vector in $\{0,1\}^{|A|}$. If the restriction of R to A is the set of all functions from A to $\{0,1\}$, then we say that R *shatters* the set A.

Definition 2.4 (VC Dimension)

The VC dimension of a set R of rules denoted by $VCDim(R)$, is the maximal size of the sets $A \subset D$ that can be shattered by R. If R can shatter sets of arbitrarily large size, we say that R has an infinite VC dimension. □

The following theorem gives the characteristics of PAC learnability and the sample complexities of sets R of rules; see e.g. [51].

Theorem 2.2.

A set R of rules is PAC learnable *if and only if* its VC dimension is finite. Moreover, if a set R of rules has a finite VC dimension d, then there are absolution constants C_1 and C_2 such that

$$C_1 \frac{d+\log(1/\delta)}{\epsilon} \leq m_R(\epsilon, \delta) \leq C_2 \frac{d\log(1/\epsilon)+\log\left(\frac{1}{\delta}\right)}{\epsilon}. \quad \square$$

The following are the VC dimensions of some example sets of rules/functions.

Table 1. Examples of VC-Dimensions

Set of Rule/Functions	**VC-Dim**
Threshold functions over the set $\mathbb{R}$ of real numbers	1
Intervals over the set $\mathbb{R}$ of real numbers	2
Axis aligned rectangles over the set $\mathbb{R}^2$ of two dimensional space of real numbers	4
Finite set H of functions	$\leq log_2(\|H\|)$
$\{\lceil 0.5\sin(\theta x)\rceil \| \theta \in \mathbb{R}\}$	∞

As shown in Table 1, any finite set of rules is PAC learnable because its VC dimension is finite. The following inequality about the sample complexity of finite set of rules is useful; see e.g. [51].

Theorem 2.3.

If a set R of rules is a finite set, its sample complexity has the following property.

$$m_R(\epsilon, \delta) \leq \left\lceil \frac{\log\left(\frac{|R|}{\delta}\right)}{\epsilon} \right\rceil. \quad \square$$

In the next subsection, we give a brief introduction to ML techniques.

2.2 ML Techniques

By ML techniques, we meant various algorithms of inductive inference devices and their implementations in ML software tools. We start with a brief discussion of various paradigms of machine learning, which differs from each other in terms of how data (i.e. instances of the target rule) are obtained and presented to the learning device. Then, we introduce two most commonly used ML algorithms: (a) artificial neuron networks, and (b) clustering algorithms.

2.2.1 ML Paradigms

There are four different paradigms of ML.

- *Supervised learning*.

In this paradigm, a ML device is provided with instances of the rule to learn. For a rule $r \subseteq D \times C$ with domain D and codomain C, the input to the inductive inference device will be a set of elements $< (a_i, b_i), l_i >, i = 1, 2, \ldots, k$, where $a_i \in D$, $b_i \in C$, and l_i is called the label for the instance (a_i, b_i), which is either *true* or *false* for the pair $(a_i, b_i) \in r$ or not. Supervised learning is applicable to learn functions and relations.

- *Unsupervised learning*.

For unsupervised learning, no labels like l_i in the supervised learning on the data are provided. Instead, a notion of similarity between the observed data is defined for, or assumed known to, the inference device. Such a notion of similarity is often defined in the form of a distance measure between the data, which is a function $\| x, y \|$ from pairs of elements x and y in the space S of examples to positive real numbers $r \in \mathbb{R}^+$. A distance function that satisfies the following properties is called a *metrics*.

(1) $\forall x, y \in S. \|x, y\| \geq 0$;

(2) $\forall x, y \in S. \|x, y\| = 0 \iff (x = y)$;
(3) $\forall x, y \in S. \|x, y\| = \|y, x\|$;
(4) $\forall x, y, z \in S. \|x, y\| \leq \|x, z\| + \|z, y\|$.

If condition (2) above is replaced by the following condition, it is called a *pseudometrics*.

(5) $\forall x \in S. \|x, x\| = 0$.

The set of examples in an unsupervised learning is often in a multiple dimensional space, where each dimension represents a feature of the element. The most common unsupervised learning methods is cluster analysis, which aims at discovering the hidden patterns in data by grouping elements similar to each other into classes. Clustering methods will be discussed in Section 2.2.3.

- *Semi-supervised learning*.

In this type of learning process, a part of the input to the inductive inference machine is labelled as in supervised learning, but some are not labelled.

- *Reinforcement learning*.

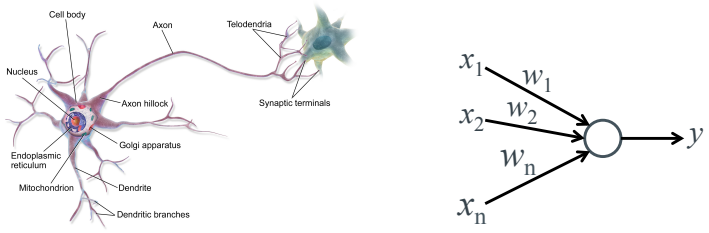
In a reinforcement learning process, an inductive inference device (often called the learning *agent* in the literature of reinforcement learning) engages in repeated sequential interactions with an environment. At each time moment t, the agent receives an observation o_t on the environment, and takes an action a_t of its choice from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1}, and a reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The agent aims at collecting as much reward as possible in the process of interactions with the environment. When the agent's performance is compared to that of an agent that acts optimally, the difference in performance gives rise to the notion of regret. In order to act near optimally, the agent must reason about the long-term consequences of its actions (i.e., maximize future income), although the immediate reward associated with this might be negative. Thus, it is well-suited to problems that include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, and play games such as checkers and go.

2.2.2 Artificial Neural Networks

An *artificial neural network* (ANN), or simply *neural network* (NN), is a network of computing nodes that are *artificial neurons* inspired from *neurons* (i.e. *nerve cells*).

- *Artificial Neurons*

As shown in Figure 2, each artificial neuron has a number of numerical inputs $x_1, \ldots, x_n$ and a numerical output y. The output is calculated by a formula $y = \varphi(\sum_{i=1}^{n} w_i x_i)$, where for $i = 1, \ldots, n$, w_i are called the *weights*, and φ is called the *transfer function* or *activation function*, which produces the value transferred from one neuron cell to the others that connected to the neuron's output as their inputs.



(a) Nerve cell (b) Artificial neuron

Figure 2. Neuron Cell and Artificial Neuron

In most neuron networks, the input to a neuron may have a bias value b in addition to the inputs such that $y = \varphi(b + \sum_{i=1}^{n} w_i x_i)$. In this case, we regard the bias value as from a constant input $x_0 = 1$, called the *bias input,* with the weight $w_0 = b$ equals to the *bias value*. Thus, the function can also be represented in the form of

$$y = \varphi\left(\sum_{i=0}^{n} w_i x_i\right). \qquad (*)$$

There are a few different types of transfer functions studied in ANN. The most commonly used ones are:

- *Step functions*: $\varphi(x) = \begin{cases} 1; & x \geq \theta \\ 0; & x < \theta \end{cases}$, where θ is the threshold.
- *Sigmoid functions*: These are S-curved functions. For example, the following logistic function is a sigmoid function.

$$\varphi(x) = \frac{1}{1 + e^{-x}}.$$

There are a few such sigmoid functions that have been studied in the ANN literature. Figure 3 shows the curves for a few often used ones.

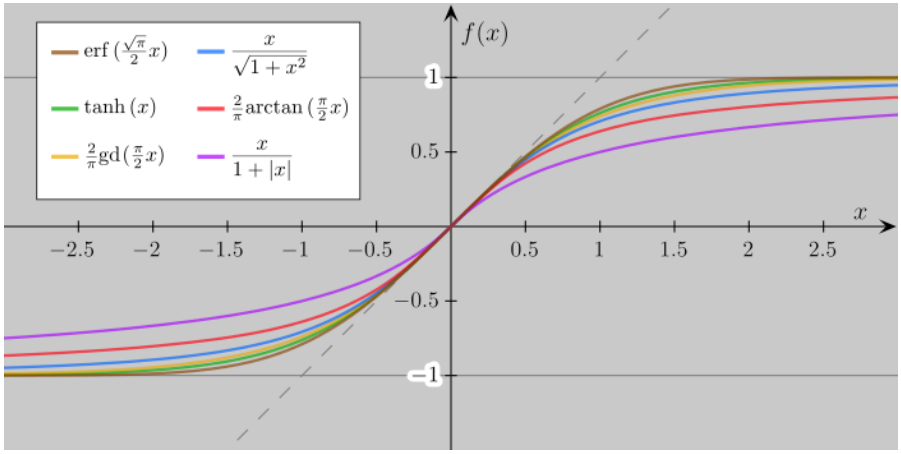


Figure 3. S-Curves of Commonly Used Sigmoid Functions

- *Rectifier function*: The rectifier function is defined as follow.

$$\varphi(x) = x^+ = Max(0, x).$$

It is also known as the *ramp* function. A smooth approximation of the rectifier function is the *analytic function* $\varphi(x) = log\,(1 + e^x)$, which is also called the *softplus* function; see Figure 4 for the curves of rectifier and softplus functions.

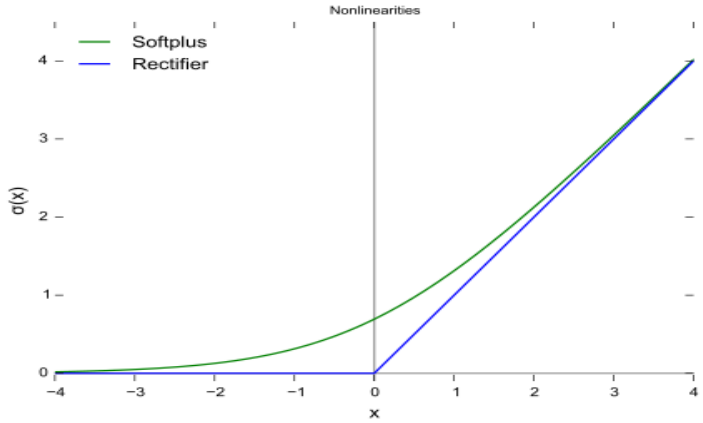


Figure 4. The Curves of Rectifier and Softplus Functions

Note that the derivative of the softplus function is the logistic function. The rectifier was introduced by Hahnloser *et al*. in 2000 as an activation function for neural networks [30]. It was demonstrated to enable

better training for deep neural networks in 2011 [61]. It is currently the most popular in the study and application of deep learning ANN. A node in ANN employs the rectifier is called a *rectified linear unit* (*reLU*). There are a few variants of ReLU in the literature of ANN. A thorough review of them is beyond the scope of this paper.

- *Types of Neural Networks*

As shown in Figure 5, a neural network typically consists of an *input layer* of neurons that takes inputs, an *output layer* of neurons that gives the outputs, and in between, a number of *hidden layers*. When the number of hidden layers in a neural network is large, typically from 5 to 20 or above, it is called a *deep neural network*.

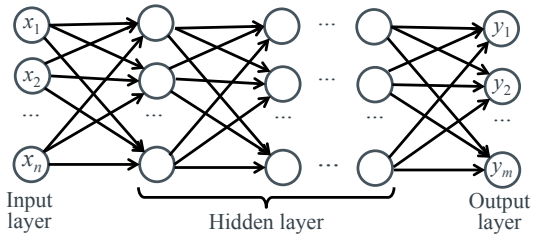


Figure 5. Multilayer Artificial Neural Networks

In general, a neural network does not need to be fully connected between two layers as in Figure 5, and these layers may have different numbers of neurons, even using different activation functions. When a neural network consists of multiple non-linear layers of different sizes, it is called *convolutional*, which is proven to be successful to be trained for image recognition. A neural network is traditionally an acyclic directed graph, but *recurrent* nets that allow cycles have been developed to process sequential inputs like text and for speech recognition.

- *Forward Propagation and Backpropagation*

For a neural network shown in Figure 5, given the inputs values $x_1, \dots, x_n$ fed into the input layer, the neurons calculates the output and feed forward to the next layer according to formula (*) and so forth until the results is propagated to the output layer. This process is called *forward propagation*.

Therefore, a neural network can be mathematical represented as a function $f_W(x_1, \dots, x_n) = (y_1, \dots, y_m)$, where $W = (\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)$, for $i = 0, 1, \dots, k$, $\boldsymbol{w}_i$ is the $n_i \times n_{i+1}$ matrix of the weights between neurons of layer i and neurons of layer $i+1$, n_i is the number of neurons of layer i. In particular, the element $w_{u,v}^{(i)}$ on row u column v of matrix $\boldsymbol{w}_i$ is the weight of the link from neuron u of layer i to neuron v of layer $i+1$.

Backpropagation is the most popular training algorithm for artificial neural networks. It is a supervised learning process with a set of input-output pairs of the required function as the training data. It starts with an initialization of the network with random weights assigned to the edges between the neurons. For each pair of input-output $(x_1, \dots x_n) \to (y_1, \dots, y_m)$ in the training data, the learning process consists of two stages: *forward phase* and *backward phase*.

The forward phase is a forward propagation of the input vector $(x_1, \dots x_n)$ through the network as a cascade of computation across the layers using the current weights W associated to the links until it reaches the output layer. Thus, the network computes the output $(y'_1, \dots, y'_m) = f_W(x_1, \dots, x_n)$. The output $(y'_1, \dots, y'_m)$ of the network is then compared to the expected output $(y_1, \dots, y_m)$ of the training data, and their discrepancy defines the error vector $(e_1, \dots, e_m) = (y'_1 - y_1, \dots, y'_m - y_m)$. The error shows whether the network has learned to compute the output correctly.

In the second stage, i.e. the backward phase, the output error propagates backward to update the weights associated to the links between neurons. The most popular training algorithm to update the weights is the *gradient decent* method, which can be understood as a solution to the optimization problem to minimize the cost of errors defined by a loss function $E(e_1, \dots, e_m)$ on the errors. A typical loss function is the square sum of errors:

$$E(e_1, \dots, e_m) = \frac{1}{2}\sum_{i=1}^{m} e_i^2 = \frac{1}{2}\sum_{i=1}^{m} (y_i' - y_i)^2.$$

By the gradient descent method, the training of the network is to adjust the weights by using the gradient to calculate the steepest descent direction. The gradients of weights associated to the links in the neural network can be calculated from the output layer backwards layer by layer until the input layer according to a recursive formula derived from the activation function associated to the neurons.

Given the gradients of the weights and the output of the neuron on the training data, the weights are updated by subtracting the old weight by the multiplication of the gradient to the old weight and a percentage, which is call the *learning rate*. If the learning rate is too small, it will take a large number of iterations of the training to converge, while if it is too large the training may results in a suboptimal solution.

The training process goes through each example in a set of training data one by one and updates the weights. One cycle of this training is called an *epoch*. One epoch may not result in a network that has acceptable error rate. Therefore, the training process may need to go through a number of epochs until either the training converges (i.e. the modifications to the weights are very small) or the error is acceptable. The training may fail, if after a large number of epochs the training still does not converge, or it converges to a network that the error is larger than acceptable.

It is worth noting that training a neural network using backpropagation implements the so-called *empirical risk minimization* (ERM) rule, i.e. to minimalize the average of the loss function on training data.

- *Theoretical Properties of Neural Networks*

We use $\langle V, E, \varphi \rangle$ to denote such a neural network that consists of a set V of neurons and a set E of connecting edges between the neuron, and φ is the activation function on all the neurons in the network. Given a mapping $\omega: E \to \mathbb{R}$ from the set E to real numbers $\mathbb{R}$, i.e. the weights associated to the links between neurons, the neural network computes a function $h_{V,E,\varphi,\omega}: \mathbb{R}^n \to \mathbb{R}^m$, where n is the number of neurons on the input layer, and m is the number of neurons on the output layer. The set of rules that can be represented by such a neural network is, therefore,

$$H_{<V,E,\varphi>} = \{h_{V,E,\varphi,\omega} | \omega: E \to \mathbb{R}\}.$$

The following theorem gives the expressiveness of neural network. It states what kinds of functions are implementable by neural networks.

Theorem 4.

(1) For every natural number n>0, there exists a neural network $\langle V, E, sign \rangle$ of depth 2 such that $H_{<V,E,sign>}$ contains all n-ary Boolean functions, i.e. functions from $\{0,1\}^n \to \{0,1\}$.

(2) The neural network $\langle V, E, \varphi \rangle$ that contains all functions of n-ary Boolean functions has the property that $|V|$ is exponential in n. □

The following theorem gives the sample complexity of neural networks. The sample complexity determines how many different training data are required to train a neural network.

Theorem 5. Let $\boldsymbol{\sigma}$ be the sigmoid function.

(1) If the weights associated to the links can be any real numbers, the VC-dimension of $H_{<V,E,\sigma>}$ is lower bounded by $\Omega(|E|^2)$ and upper bounded by $O(|V|^2|E|^2)$.

(2) If the weights associated to the links are b bits of floating point numbers, the VC-dimension of $H_{<V,E,\sigma>}$ is $O(b|E|)$. □

The following theorem gives the computational complexity to implement ERM. It indicates how many epochs are needed to train a neural network.

Theorem 6.

It is NP hard to implement the ERM rule using neural network $\langle V, E, sign \rangle$ which contains n input neurons,

and a single hidden layer that contains at least 3 neurons. □

2.2.3 Clustering

Clustering is a ML technique widely used in data mining to uncover hidden rules in the data. The input to a clustering algorithm is a set $D = \{x_1, x_2, \dots, x_n\}$ of unlabeled data, where each element x_i is a value in a k-dimensional space $D_1 \times D_2 \times \dots \times D_k$. Typically, each dimension represents a feature of the elements, and a distance function $\|.,.\|_i : D_i \times D_i \to \mathbb{R}^+$ is defined on the values of the feature. The distance between two points in a k-dimensional space D can be calculated from the distances on each dimension, for example, using *Euclidean* distance defined as follows.

$$\delta(\boldsymbol{x}, \boldsymbol{y}) = \sqrt[2]{\sum_{i=1}^{k} \|x_i, y_i\|_i}$$

where $\boldsymbol{x} = \langle x_1, \dots, x_d \rangle$ and $y = \langle y_1, \dots, y_d \rangle$. However, one may use other distance functions as far as it satisfies the properties of metrics or pseudometrics.

A clustering $\boldsymbol{C} = \{C_1, C_2, \dots, C_k\}$ is a partition of the data set D. That is, C must have the following properties.

- Each cluster C_i is a non-empty subset of data in D, i.e. $\forall i = 1, \dots, k. (\emptyset \subset C_i \subseteq D)$;
- The clusters are pairwise disjoint, i.e. $\forall i \neq j \in \{1,2, \dots, k\}. (C_i \cap C_j = \emptyset)$;
- The clusters cover all elements of the data set D, i.e. $\bigcup_{i=1}^{k} C_i = D$.

The goal of clustering is to partition the data set into k clusters that each cluster contains elements that are similar to each other (i.e. have a small distances from each other) and elements from different clusters are less similar to each other (or have a larger distance between them). There are a number of clustering algorithms available for different types of data spaces. Among them, one of the most frequently used clustering techniques is *hierarchical clustering*, which have been applied in the research on software testing; see, for example, [20].

Given a data set $D = \{x_1, x_2, \dots, x_n\}$, a clustering $\boldsymbol{A} = \{A_1, A_2, \dots, A_r\}$ is nested in clustering $\boldsymbol{B} = \{B_1, \dots, B_s\}$, if for each $A_i \in \boldsymbol{A}$, there is a $B_j \in \boldsymbol{B}$ such that $A_i \subseteq B_j$. Hierarchical clustering yields a hierarchical structure of clustering, or formally, a sequence $\boldsymbol{C}_1, \boldsymbol{C}_2, \dots, \boldsymbol{C}_m$ of clusterings such that for each $i = 1, \dots, m-1$, clustering $\boldsymbol{C}_i$ is nested in clustering $\boldsymbol{C}_{i+1}$, and the sequence of clusterings ranges from $\boldsymbol{C}_1 = \{\{x\} | x \in D\}$ to $\boldsymbol{C}_m = \{D\}$. Figure 6 shows an example of hierarchical clustering of a set of 5 elements {a, b, c, d, e}.

Clustering	Clusters
$\boldsymbol{C}_1$	{a}, {b}, {c}, {d}, {e}
$\boldsymbol{C}_2$	{a}, {b, c}, {d}, {e}
$\boldsymbol{C}_3$	{a}, {b, c}, {d, e}
$\boldsymbol{C}_4$	{a}, {b, c, d, e}
$\boldsymbol{C}_5$	{a, b, c, d, e}

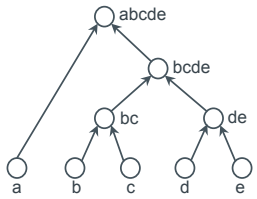


Figure 6. Example of Hierarchical Clustering

There are two main approaches to generate hierarchical clustering: *agglomerative* and *divisive*.

The *agglomerative* approach works bottom-up starting from the finest clustering $\boldsymbol{C}_1 = \{\{x\} | x \in D\}$. Each step it merges two most similar clusters until it reaches the most coarse cluster $\boldsymbol{C}_m = \{D\}$.

The *divisive* approach starts from the most coarse clustering $\boldsymbol{C}_m = \{D\}$ and works top-down. At each step, it spilt one cluster into two until it reaches the finest clustering $\boldsymbol{C}_1 = \{\{x\} | x \in D\}$.

Agglomerative approach is more suitable for analysing numerical data, while divisive approach is more efficient for analysing graphs. The key step in the agglomerative algorithms of hierarchical clustering is to calculate the distances between a pair of clusters. Let $\delta(x, y)$ be a distance function between two points in the d-dimensional space D. The following distance functions between two clusters are often used in

clustering. We will use $\delta(X,Y)$ to denote the distance between two clusters *X*, and *Y*.

- *Single Link*: The distance between two clusters *X* and *Y* is the minimal distance between a point *x* in *X* and a point y in *Y*. Formally,

$$\delta(X,Y) = min\{\delta(x,y)|x \in X, y \in Y\}.$$

- *Complete Link*: The distance between two clusters *X* and *Y* is the maximal distance between a point *x* in *X* and a point y in *Y*. Formally,

$$\delta(X,Y) = max\{\delta(x,y)|x \in X, y \in Y\}.$$

- *Group Average*: The distance between two clusters *X* and *Y* is the defined as the average pairwise distances between the points in *X* and *Y*. Formally,

$$\delta(X,Y) = \frac{\sum_{x\in X}\sum_{y\in Y}\delta(x,y)}{\|X\|\times\|Y\|}$$

- *Mean Distance*: The distance between two clusters *X* and *Y* is the defined as the distances between the mean (or centroids) of the clusters *X* and *Y*. Formally,

$$\delta(X,Y) = \delta(\mu_X,\mu_Y)$$

  where $\mu_X = \frac{1}{n}\sum_{x\in X} x$.

- *Minimum Variance* (*Ward's Method*): The distance between two clusters is defined as the increase in the sum of square errors (SSE) when the two clusters are merged. Formally, the sum of square errors *SSE*(*X*) of a cluster *X* is defined as follows,

$$SSE(X) = \sum_{x\in X}\|x-\mu_X\|^2.$$

Then, the distance between clusters *X* and *Y* is defined as follows.

$$\delta(X,Y) = \Delta SSE(X,Y) = SSE(X\cup Y) - SSE(X) - SSE(Y).$$

Note that $\Delta SEE(X,Y) = \left(\frac{\|X\|\times\|Y\|}{\|X\|+\|Y\|}\right)\|\mu_X-\mu_Y\|^2$. Therefore, minimum variance is actually a weighted mean distance measure if we use Euclidean distance.

The computational complexity of agglomerative clustering is $O(n^2\log n)$, where *n* is the number of elements in the data set to be analysed. Note that to achieve a perfect clustering is NP hard. Agglomerative clustering does not guarantee to general a perfect clustering.

3. Test Adequacy

The test adequacy problem is concerned with how to measure the thoroughness of a software test and how to determine when a test can stop. It has been intensively investigated since Goodenough and Gerhart introduced the notion of test adequacy criteria in 1976 [28]. Several dozens of test adequacy criteria have been proposed and studied in the literature; see, e.g. [69] for a survey of unit test adequacy criteria. Some of them are widely used in software testing practices. However, how to measure test adequacy is still an open problem in software testing research. In particular, despite of the large number research papers on test adequacy criteria published in the literature, we still cannot claim that the software is correct or reliable if it passes an adequate test successfully. In this section, we review how ML theories and techniques are employed to solve research problems related to test adequacy.

3.1 The Notion of Test Adequacy

A test adequacy criterion can be defined as a stop rule or a measurement. The former can be formally defined as a predicate on test sets and the software under test, which consists of a program and a specification [69]. The later can be defined as a mapping from test sets and software to a real number in the unit interval.

Definition 3.1 (Test Data Adequacy Criteria as Stopping Rules)

A test data adequacy criterion C is a function $C: P\times S\times T \to \{true, false\}$. $C(p,s,t) = true$ means that

the test set t is adequate for testing program p against specification s according to the criterion C, otherwise t is inadequate. □

Definition 3.2 (Test Data Adequacy Measurements)

A test data adequacy measurement M is a function $M: P \times S \times T \rightarrow [0,1]$. $M(p,s,t) = r$ means that the adequacy of testing the program p by the test set t with respect to the specification s is of degree r according to the criterion M. The greater is the real number r, the more adequate is the testing. □

In practice as well as in research, a test adequacy criterion is often represented as a test case design rule for the generation of test cases, which can be formally defined as follows.

Definition 3.3 (Test Data Adequacy Criteria as Design rules)

A test data adequacy criterion C is a function $C: P \times S \rightarrow \mathcal{P}(D)$, where D is the input domain of the software, $\mathcal{P}(D)$ is the power set of D, i.e. the set of all possible test sets. A test set $t \in C(p,s)$ means that t is an adequate test set for testing program p against specification s according to the criterion C, otherwise t is inadequate. □

Test adequacy criteria as design rules are mathematically equivalent to as stop rules.

There are a large number of test adequacy criteria proposed and studied in the literature. As discussed in [69], there are a number of different ways to classify these criteria. One of the ways is according to the basic ideas underneath the testing method and which leads to the following types of test adequacy criteria.

- *Structural test criteria*, such as statement coverage, branch coverage, path coverage, etc. in control flow testing, and definition-use path coverage, definition context coverage, etc. in data flow testing.
- *Fault-based test criteria*, such as mutation score;
- *Error-based test criteria*, such as boundary coverage in partitioning testing and functional adequacy in Howden's algebraic testing.

Since 1980s, ML techniques and theories of inductive inferences have been applied to define the ideal test adequacy criteria, to analyze the weakness of existing ones, and more recently, to develop prototype tools to measure test adequacy. The following subsections review these works.

3.2 Weyuker's Inference Adequacy Criterion

In search for an ideal test adequacy criterion, Weyuker (1983) proposed an adequacy criterion explicitly employing ML [58]. As shown in Figure 7, Weyuker's idea is that, given an inductive inference tool that can generate a rule from examples, a test is defined to be adequate if the rule generated from the test cases is equivalent to both the program under test and its specification.

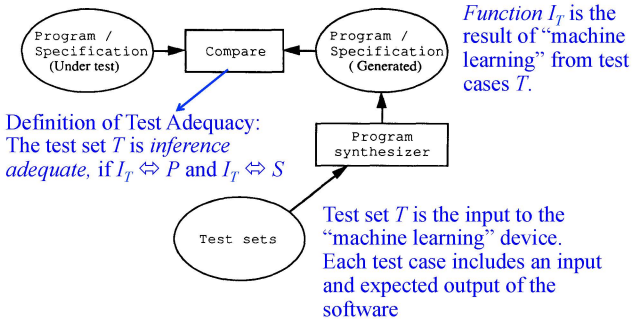


Figure 7. Weyuker's Inference Adequacy [58]

Definition 3.4 (Inference Adequacy Criterion)

Let p be a program under test and s be its specification, $t = \{< x_i, y_i > | i = 1,2,\ldots,n\}$ be a set of test cases where x_i are the inputs and y_i are the corresponding expected outputs, and M be a program synthesis device. The test set T is defined to be adequate according to the *inference adequacy* criterion, if $p \Leftrightarrow M(t)$ and $s \Leftrightarrow M(t)$. □

Weyuker proved the following properties of inference adequacy.

Theorem 3.1 (Weyuker Theorem)

Let p be a program intended to fulfil specification s. Then, we have that:

(1) If program p can be inferred from test set t, then t is branch adequate for p, i.e. the coverage of all branches of program p, if p has no dead code.

(2) If test set t is inference adequate for p relative to s, then t is an ideal test set for p, i.e. we can guarantee the correctness of program p w.r.t. s if p passes on all test cases in t.

(3) There exists a test set t that is ideal for p, but is not inference adequate for p relative to s. □

In the paper [58], Weyuker analysed the practical limitations of inference adequacy. Summer's program synthesis system [53], from which Weyuker drawn an example to illustrate how inference adequacy would work, is very limited in the power to generate complex programs. Program synthesis techniques were actively developed in the 1980s and 1990s and improved on the power to generate more complicated programs from examples than Summer's work; see for example, [34],[52],[64],[65]. However, the powers of such systems are still very limited even today; see for example, [41]. Weyuker considered plausible approximations to the inference adequacy criterion. She pointed out that restriction on a subset of programs, such as finite state machines "*is not likely to be productive*." However, relaxations on the conditions that the inference result is equivalent to both program and the intended specification could be useful. In particular, if the test set is generated from specification, one could use "*program-adequate*", i.e. the test is adequate if the program is equivalent to the derived rule. If the test set is generated from program, one could use "*specification-adequate*", i.e. the derived rule is equivalent to the specification.

Weyuker is the first person who applied a ML technique to software testing. Her pioneering work is of greater theoretical value than practice. First of all, equivalence between a generated rule and a program/specification is undecidable in general. Second, if one can prove that the function generated from a set of test cases (including inputs and the expected outputs) is equivalent to both the program and the specification, it actually proved indirectly that the program is equivalent to the specification. There seems no reason why the generated function is easier to prove its equivalence to the program or the specification than to prove the program is equivalent to the specification directly.

Weyuker also pointed out that "*it would be both interesting and useful to attempt to develop practical approximations to program inference*" [58]. One form of practical approximation approach to program synthesis is Valiant's PAC ML protocol proposed in 1984, where PAC stands for *Probably Approximately Correct* [54]. It does not require the generated rule to be absolutely correct, but tolerant the error as far as the probability of error occurrence is small enough. This approach is taken by Zhu *et al*. in the 1990s and by Fraser *et al*. more than 30 years later; see subsections below.

3.3 Testing as Inductive Inference

Inspired by Weyuker's work, Zhu, Hall and May compared the structure of software testing processes with inductive inference processes and made two observations [68].

First, software testing is actually an inductive inference process in the course of which one observes the program's behaviour on a finite subset of samples and attempts to draw a general conclusion on the whole behaviour space.

Second, test adequacy criteria play the same role as the convergence criteria in inductive inference process [68]. A question is, therefore, whether those test adequacy criteria proposed in the literature and used in practice are actually rules for determining the convergence of inductive inference underlying the testing process.

Unfortunately, there is no evidence to believe that practically used test adequacy criteria are convergence rules of the inductive inference underneath testing process. Instead, these test adequacy criteria reported in the literature are approximations of the so-called ideal test adequacy criteria. Thus, the question becomes: whether or not the so-called ideal test adequacy criteria are actually convergence criteria of inductive inference? Zhu addressed this question in [66] by proving that Weyuker's inference adequacy criteria satisfy the axioms that characterises the ideal test adequacy criteria.

There are four axiom systems of test adequacy criteria reported in the literature.

- Weyuker's axiom system on program based test adequacy criteria; see Table 2. It is the first set of axioms of test adequacy criteria. It was proposed in [59] and extended in [57] by Weyuker, formalised and analysed by Parrish and Zweben [41][43], Hamlet [31], and Zweben and Gourlay [72].
- Zhu's axiom system on control flow test adequacy criteria [67]. It extends Weyuker's axioms and adapts Baker *et al*'s required properties of control test adequacy criteria [10];
- Miao and Liu's axiom system on predicate coverage criteria. It is an adaptation of Zhu's axioms of control flow adequacy criteria to predicate coverage criteria [39].
- Zhu and Hall's axiom system on test adequacy measurements [63]; see Table 3.

Table 2. Weyuker Axiom System of Program-Based Test Adequacy Criteria

Name	**Meanings**
Finite Applicability	For every program, there exists a finite adequate test set.
Non-exhaustive Applicability	There is a program p and a test set t such that p is adequately tested by t and t is not an exhaustive test set.
Monotonicity	If test set t is adequate for testing program p and $t \subseteq t'$, then test set t' is also adequate for testing p.
Inadequate Empty Set	The empty set is not adequate for any program.
Anti-extensionality	There are semantically equivalent programs p and q such that there is a test set t that is adequate for testing p, but not adequate for testing q.
General Multiple Change	There are programs p and q which are of the same shape and a test set t such that t is adequate for testing p, but not adequate for testing q.
Anti-decomposition	There exist a program p, its component q and a test set t such that t is adequate for testing p, but test set t' is not adequate for testing q, where t' is the set of vectors of values that variables can assume on entrance to q for some test cases x in t.
Anti-composition	There exist programs p and q and test sets t such that t is adequate for testing p and $p(t)$ is adequate for testing q, but t is not adequate for $(p;q)$, where $(p;q)$ is the sequential composition of p and q.
Renaming Property	Let p be obtained by systematically renaming some variables in q. Then, for all test sets t, t is adequate for testing p if and only if t is adequate for testing q.
Complexity property	For every natural number $n>0$, there is a program p, such that p can be adequately tested by a size n test set, but not by any size $n-1$ test set.
Statement Coverage Property	If a test set t is adequate for testing p, then, every feasible statement in p will be executed when p is tested on t.

Zhu formally defined the notion of adequacy criterion induced from an inductive inference device as follows [66].

Definition 3.5 (Program-based adequacy criterion induced from an inductive inference device)

Let M be any given inductive inference device. An adequacy criterion as a stop rule induced from inductive inference device M, written $C_M(t,p)$, it is the adequacy criterion such that for all programs p and test sets t, $C_M(t,p) \Leftrightarrow M(p \downarrow t) = p$, where t is a finite test set for program p, $p \downarrow t$ is the subset of input/output pairs of p with input from t. □

Using properties of *Identification in the limit*, Zhu proved that the adequacy criterion induced from an inductive inference device is a valid interpretation of the main axioms in the Weyuker system of program based adequacy criteria. Let M be an inductive inference device and $C_M(t,p)$ be the adequacy criterion induced from M as defined by Definition 3.5. Formally,

Theorem 3.2 (Zhu 1996)

The adequacy criterion $C_M(t,p)$ for a set of programs P satisfies the Weyuker's axioms of *finite*

applicability, *monotonicity*, *inadequacy of the empty test set*, *complexity property*, and *statement coverage property* for program-based test adequacy criteria, if M has the following properties.

(1) M is conservative, which means that M changes its output hypothesis only if the hypothesis is not consistent with a new input instance.

(2) M has the simplest hypothesis property.

(3) P is explanatorily learnable by M in the identification in the limit.

□

The original motivation of Weyuker's inference adequacy criterion is to ensure the correctness of the program if it passes an adequate test. The following theorem proved in [66] shows that this is achievable without actually prove the equivalence between programs and without actually generate a program from the test cases. The condition for the correctness is the learnability of the program and the specification by the inductive inference device.

Theorem 3.3 (Zhu 1996)

A program p is correct *w.r.t.* a specification s after successfully tested on a finite test set t, if t is adequate according to criterion $C_M(t,p)$, p is explanatorily learnable by M, s is behaviourally learnable by M, and M converges to a function that is consistent with s on t. □

Zhu also explored the possibility of using PAC learning to define test adequacy measurements. The following definition was given in [66].

Definition 3.6 (Adequacy measurement induced from an inductive inference device)

Let P be a set of programs and M be an inductive inference device such that $m_P(\epsilon,\delta)$: $(0,1)^2 \to \mathbb{N}$ is the sample complexity of P for M. The function $K_{M,\delta}(t,p)$ from test sets t and programs p in P to real numbers in the unit interval [0,1] is defined as follows.

$$K_\delta(t,p) = 1 - sup\,\{\epsilon | m_P(\epsilon,\delta) \geq ||t||\}$$

where $sup(\emptyset) = 0$. The function $K_{M,\delta}(t,p)$ is called the *adequacy measurement* for programs in P induced from inductive inference device M and sample complexity function $m_P(\epsilon,\delta)$. □

It was also proved that if the set of programs P is PAC learnable by M with sample complexity function $m_P(\epsilon,\delta)$, the adequacy measurement induced from $(M, m_P(\epsilon,\delta))$ is a valid model of Zhu and Hall axiom system of test adequacy measurement. These axioms are given in Table 3.

Theorem 3.4 (Zhu 1996)

If a set of programs P is PAC learnable by M with sample complexity function $m_P(\epsilon,\delta)$, for any given real number $0 < \delta < 1$, $K_\delta(t,p)$ defined by Definition 3.6 satisfies the axioms of Zhu and Hall axiom system of test adequacy measurements. □

Table 3. Zhu, Hall and May's Axiom of Program-Based Test Adequacy Measurement

Name	Meanings	Formal definition
Inadequacy of an empty test	The empty set is inadequate as a test set for all software.	$\forall p \in P. M(p,\emptyset) = 0$
Adequacy of exhaustive testing	The exhaustive test set is adequate for all software.	$\forall p \in P. M(p,D) = 1$, where D is the input domain of program p.
Monotonicity	The more test cases are used, the more adequate the test.	$\forall p \in P. \forall t,t'. (t \subseteq t' \Rightarrow M(p,t) \leq M(p,t'))$
Law of diminishing returns	The more a program has been tested, the less a test set can further contribute to test adequacy in the context.	$\forall p \in P. \forall t,u,v \subseteq D.$ $(u \subseteq v \wedge t \cap u = \emptyset \wedge t \cap v = \emptyset \Rightarrow$ $M(p,v\|t) \leq M(p,u\|t))$, where $M(p,x\|t) = M(p,x \cup t) - M(p,x))$

Convergence	Take a sequence (even infinitely many) of test sets and measuring test adequacy for each step is equal to measuring the adequacy of the overall test set.	$\forall p \in P. \forall t_1, \ldots, t_n, \ldots .$ $(\lim_{n\to\infty} M(p, t_n) = M(p, t_\infty))$, where $t_\infty = \bigcup_{n=1}^{\infty} t_n)$.
Finite applicability	An adequacy criterion is finitely applicable if, for all degrees of adequacy less than 1, there always exists a finite set of test cases that achieves the required adequacy degree.	$\forall p \in P. \forall r < 1. \exists t. (M(p, t) \geq r \wedge \|\|t\|\| < \infty)$

Analysing the relationship between PAC learnability and software reliability, Zhu introduced a new concept about software reliability: *probable reliability*.

Definition 3.7 (Probable reliability)

Assume that a software system p is tested on a set t of test cases selected at random independently according to an identical distribution Pr. If the probability that the failure rate of p on t is less than or equal to ε is at most δ, we say that the δ-*probable reliability* of the software is ε. □

Probable reliability is an extension of Hamlet's concept of *probable correctness* [32]. Based on the notion of probable reliability, Zhu further explored how to take software complexity into consideration in the assessment of software reliability rather than simply treating software as a black box [71]. In fact, the probable reliability of a program that passes a test can be estimated according to the adequacy measurement induced from an inductive inference machine if the program and specification is PAC learnable.

Theorem 3.5 (Zhu 1996)

For a finite random test set t, if the program p is correct on test cases in t w.r.t specification s, and p and s are in a set R of functions that are PAC learnable, then the δ-probable reliability of program p with respect to specification s is $K_\delta(t, R)$. □

It is worth noting that, firstly, the assertion on the test adequacy measurement of a random test and the assessment of probable reliability according to the above theorem does not require the program or the specification to be actually generated by the inductive inference device. Secondly, $K_\delta(t, R)$ depends on the sample complexity of the set R of rules, which can be determined by the VC dimension of the set R. Thus, probable reliability of the software that passes a random testing depends on the complexity of the software.

3.4 Fraser and Walkingshaw's Behavioural Adequacy Criterion

More recently, Fraser and Walkinshaw employed PAC inductive inference protocol to define the so-called *behavioural adequacy criterion*, which requires an accurate model of the software to be derivable from test cases [22], [23]. The process of testing in Fraser and Walkinshaw's approach as proposed in [23] is illustrated in Figure 8, where the arcs are numbered to indicate the flow of events.

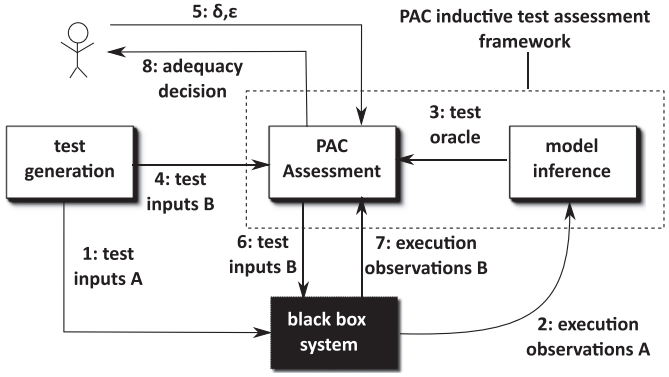


Figure 8. Fraser and Walkingshaw's Behaviour Adequacy Criterion [23]

The test generator produces a set A of test cases according to some fixed probability distribution on the input domain. They are executed on the software under test, which is treated as a black box. The test set A is also to be assessed for test adequacy. The executions of the software under test are recorded and supplied to the model inference tool to generate a hypothetical test oracle. To enable the assessment of the adequacy of test set A, the test generator produces a further test set B. The observations of the software on test cases in B are then compared against the expected observations from the model to compute the error rate. The user may supply the acceptable error bounds ε and δ. If the error rate is smaller than ε, the model inferred from observations on test set A is approximately accurate, and thus the test set can said be *approximately adequate*. The δ parameter is used to assess the confidence in the assertion of the test adequacy. By running multiple experiments on test sets B_1, B_2, …, B_n, one can count the proportion of times that the test set is approximately adequate. If, over a number of experiments, this proportion is greater than or equal to $1-\delta$, Fraser and Walkingshaw calls the test set *A probably approximately adequate*, to paraphrase the term 'probably approximately correct'.

Fraser and Walkingshaw noticed that the test cases that are behaviourally adequate cannot guarantee to cover all the statements and branched in the program under test. To overcome this problem, they employed genetic algorithms to generate test sets that cover all statements and branches in search-based approach. However, they later realised [22] that this approach potentially undermines the validity of using PAC ML theory, which requires the examples are draw at random independently using the identical probability distribution[1]. Therefore, an improved method is proposed [22], which replaced the *A*-*B* test sets by *k*-fold cross-validation, a technique widely used in data mining.

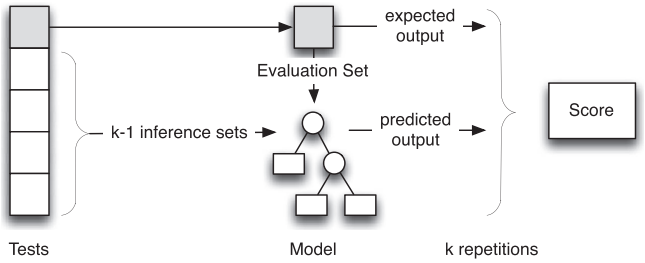


Figure 9. Illustration of Fraser and Walkingshaw's Improved Behavioural Testing Process

As illustrated in Figure 9, in the improved behavioural testing process, which is called $BESTEST_{CV}$, the test set is divided into $k > 1$ subsets. Test cases in $k-1$ subsets are used to infer the model and to calculate the average error rates on the $k-1$ test subsets, while the k'th subset is used to validate the model.

In both papers [23] and [22], experiments with testing Java programs were reported. The particular ML algorithms used in [23] are C4.5, and M5, while in [22] a wider range of ML algorithms are used, which are:

- The C4.5 Decision Tree learner for discrete systems [47]
- The M5 learner and the M5Rules variant for numeric systems [21]
- A Naive Bayesian Network learner for discrete systems [21]
- The AdaBoost learner for discrete systems [25]
- Multilayer Perceptron (neural network) learners for both numeric and discrete systems [21]
- Additive Regression for numeric systems [21]

Their experiments demonstrated that if configured properly, optimizing test generation with respect to behavioural adequacy significantly outperformed current baseline techniques in terms of fault detection, especially for larger functions with a complex branching structure. However, the subject programs used in

[1] Fraser and Walkingshaw were concerned with the independence between test cases in test sets A and B. However, in my opinion, there is another problem. That is, the test cases in A and B may be draw with different distributions if A is generated by using a genetic algorithm while B is draw at random. Therefore, the assumption of independent identical distribution made in PAC learning protocol does not hold.

the experiments are still quite small in size, range from 9 lines to 169 lines of source code, and 3 to 93 branches. The data types of the input and output of these subject programs are simple, too: numerical and/or strings. However, scalability was not considered as a potential problem [22]. Instead, complexity could be the main issue for the practical application of the technique. The time spent on inferring and evaluating the models is a small proportion of the overall testing time, which is about 7.6% on average. The accuracy of an inferred model can vary significantly, depending on a wide range of factors. Fraser and Walkinshaw pointed that it is almost impossible to answer the question of how different factors lead to the successful inference of a model (irrespective of whether the system in question is a software function). This is a well-known problem of ML techniques: Wolpert and Macready's theorem of '*no free lunch*', which states that *no learning algorithm achieves consistently better generalization performance than any other over all possible target functions* [60].

It is worth noting that, in comparison with Zhu's approach, Fraser and Walkinshaw do not consider the program under test as an element in a set of rules to learn. As we have seen in the previous subsection, if we know that the program is in a PAC learnable set of rules, it is unnecessary to actually derive the model from test cases as far as the number of random test cases is greater than the sample complexity of the set of rules for the inductive inference device.

Moreover, Fraser and Walkinshaw's behaviour adequacy criterion is a program-based adequacy criterion, which determines test adequacy solely based on the information contained in the program. Such a test adequacy criterion cannot ensure that the software has no omission error even if it passes an adequate test. That is, if the program omits a function required by the specification, a behaviourally adequate test does not guarantee to detect the error. Therefore, behaviour adequacy criterion cannot guarantee the correctness/reliability of the software under test.

4. Test Oracle

The test oracle problem is concerned with how to determine the correctness of software's output or behaviour on test cases. A test oracle is a device or mechanism that can determine the correctness of software's behaviour and/or output observed during a test execution. Often, a qualified tester plays the role of test oracle to decide whether a test execution is correct. A common practice to solve the test oracle problem is to generate test cases with expected output on the input. The actual output from the program obtained from testing is then compared against the expected output. Another approach to the test oracle problem is to employ formal specifications to check the correctness of test output; see [33] for a survey on the uses of formal specification in software testing. Test oracle problem is still one of the most costly and difficult problems in software testing.

The application of ML to test oracle can be dated to 2002 [55]. The basic idea is first to use a set of know test cases and/or software behaviours to learn a rule. And, then, for unknown test cases, the learned rule is used to determine whether the behaviour and/or output of the program are correct. Applying ML to the test oracle problem is one of the most active research topics in the recent years. A variety of approaches have been proposed and advanced in the literature. They can be classified into three categories according to the outcomes of the learning process.

- Predictors of the software's output or behaviour;
- Classifiers of the test results;
- Specifications of the software under test.

This section reviews these approaches one by one after a brief introduction to the basic concepts associated to test oracles.

4.1 Basic Concepts of Test Oracles

When a test oracle does not quarantine the correctness of its classification of test results, it may make two types of errors: *false positive* and *false negative* errors. By a *positive test*, we meant the program pass the test, while by a *negative test* we mean the program fails on the test case. A *false positive error* is the situation when the test oracle says the program pass the test while the program is actually incorrect on the test. A *false negative error* occurs when the test oracle says that the program failed on the test but the

program is actually correct on the test case. Thus, the quality of a test oracle can be measured by precision, recall, accuracy and error rate, which are defined below.

Let *T* be the total number of test cases executed, *FP* be the number of false positive test case in the test, *FN* is the number of false negative test cases, *TP* be the number of true positive, and *TN* be the number of true negative test cases.

- *Precision*: The proportion of the number of failures correctly detected by the test oracle over the total number of failures detected by the test oracle, i.e.

$$Precision = \frac{TN}{TN + FN}$$

This parameter indicates how well the tester can be sure about the bugs detected by testing are real bugs. If the precision is poor, a large amount of efforts will be wasted on investigate false bug reports.

- *Recall*: the proportion of the number of failures that are correctly detected by the test oracle over the total number of failures that should be detected by a perfect test oracle, i.e.

$$Recall = \frac{TN}{TN + FP}$$

This parameter indicates the fault detection ability of the oracle. If recall is poor, a large number of failures will not be detected and poor quality software could be released without noticing its poor reliability.

- *Accuracy*: the proportion of test cases that are correctly classified as failure or success over the total number of test cases executed, i.e.

$$Accuracy = \frac{TP + TN}{T} = \frac{TP + TN}{TP + TN + FP + FN}.$$

This parameter indicates the trustworthiness of the test oracle.

- *Error Rate*: the proportion of incorrect classification of test cases by the test oracle, i.e.

$$Error\ Rate = \frac{FP + FN}{T} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - Accuracy.$$

Now, we review various approached to ML applications to the test oracle problem.

4.2 Predictors of Software Outputs

As illustrated in Figure 10, the basic idea of this approach is to use a set of input/output pairs as the input to generate a function that is capable to predict the correct output when a new input is given. The predicted output is compared with the output produced by the software under test. If the actual output does not match the predicted output, it detects an error.

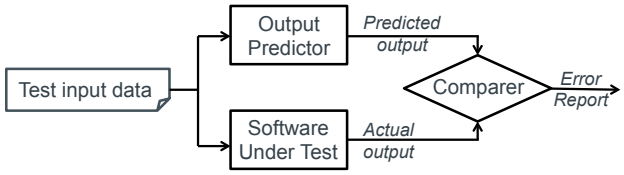


Figure 10. Use of Output Predictor as A Test Oracle

This is the approach taken by Vanmali, Last and Kandel in 2002 [55], which is the earliest work on the application of ML to test oracle problem as far as we know. As shown in Figure 11, they used backpropagation to train a neural network of three layers: the input layer, a hidden layer and the output layer.

The input layer takes input of the test case, each neuron for one input variable. The output layer presents the predictions on the output of the program. They used a 1-of-n encoding of outputs, where *n* neurons on the output layer are used to predict the value of one output variable; each neuron represents a prediction on the value of one output variable being in a given sub-range. If the output variable is a binary Boolean type

data, two neurons will be used: one for the value to be *True* and the other for the value to be *False*. If the output is a numerical value, a number of neurons will be used: each represents an interval of the possible values. The output neuron having the highest-value is activated as the *winning output*, which is taken as the network's prediction for the output of the tested program. The difference among the outputs is used as a measure of the confidence in the network prediction.

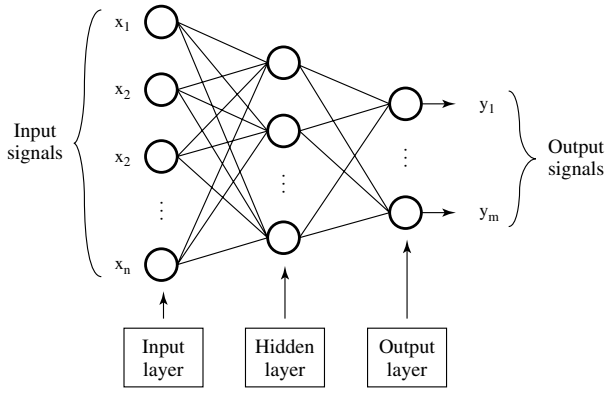


Figure 11. Structure of Vanmali, Last and Kandel's Neural Network

Vanmali, Last and Kandel reported an experiment with a credit card approval program. The program has eight input variables for information about the credit card applicant and two output variables: one binary value for whether to approve or not to approve the credit card application; one integer value for the amount of credit limit. They used a set of 500 test cases as the training data with a learning rate of 0.5, and the network required 1,500 epochs to produce a 0.2% misclassification rate (error rate) on the binary output and 5.4% for the continuous output. Figure 8 shows the number of epochs versus the convergence of the error rate.

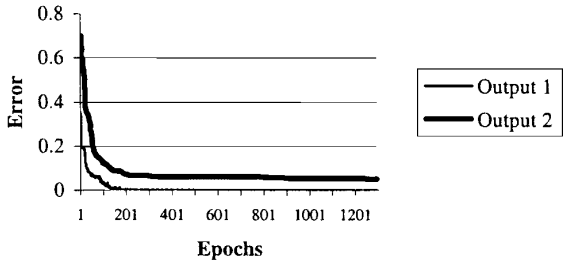


Figure 12. Error Convergence [55]

After the training of the neuron network, 21 mutated programs with injected faults were tested on 1000 test cases generated at random using the neuron network as the test oracle. The test oracle achieved the minimum average error rate of 8.31% for the binary output, and the minimum average error rate of 20.79% for the continuous output.

It is worth noting that the test oracle generated through ML is an approximation to the function of the program because they use an interval to predict the actual value of numerical output. This can significantly reduce the computational complexity of the test oracle, but consequently, higher error rate. Vanmali, Last and Kandel noticed that no matter how to adjust the ranges of the intervals represented by the neurons on the output layer, the average error rates on the numerical output variable cannot be reduced. Therefore, how to improve the misclassification rate is one of the key issues for using ML to solve the test oracle problem.

An alternative to neural network is info-fuzzy networks (IFN) [37]. An IFN is a directed rooted graph that represents a decision procedure. The classification for a given input starts from the root node and traverses the graph until reaching a target node. Supervised learning can be used to train an IFN in a way that the weights adjustment is done after the entire training set is presented to the system. Agarwal, Tamir, Last and Kandel (2012) reported a set of experiments that compare neural network and info-fuzzy network as test

oracles [1]. They demonstrated that IFN significantly outperforms the ANN in terms of computation time while providing nearly the same fault detection effectiveness.

In 2012, Shahamiri, Wan-Kadir, Ibrahim, and Hashim proposed a multi-networks approach to test oracle problem [50]. The basic idea is that for each output variable O_i of the program, a separate neural network ANN_i is used to predict the output value of the variable O_i. Therefore, if there are n output variables, the architecture of the test oracle consists of n neural networks; see Figure 13. Their case study on two example systems demonstrated that multiple network approach consistently outperforms single network approach on error rate, accuracy and fault detecting ability.

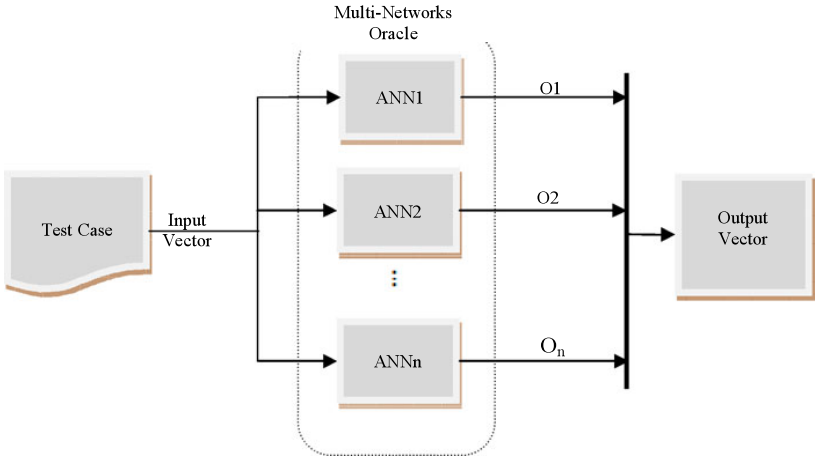


Figure 13. Architecture of Test Oracle in the Multi-Network Approach

4.3 Classifiers of Test Results

The second approach to apply ML technique to the test oracle problem is to learn a classifier that classifies observed behaviours of the software under test into correct or incorrect without generating an expected output. The observed behaviours are not limited to the input/output pairs, but more often contain information about execution paths, such as branch profiles. The observed behaviours can be labelled as *pass* or *fail* for supervised learning. When a model is trained on a set of instances such that new observed behaviours can be classified correctly into *pass* or *fail*, test results can be determined as correct or not; thus solves the test oracle problem.

Not only can such a classifier be used as a test oracle, but also useful for many other purposes in software testing and analysis. In fact, many early works on learning behaviour classifiers are not just aiming at test oracle problem. For example, Dickinson, Leon and Podgurski (2001) aims at reducing test costs in behaviour-based testing [20]. Podgurski, Leon, Francis, Masri, and Minch (2003) is concerned with debugging [44]. Bowring, Rehg and Harrold (2004) pointed out that such a classifier can be used in a number of scenarios, including test case reduction and test oracle [12]. Their uses in test case prioritization and test suite reduction will be covered in the next section.

A typical example of this type of works is reported by Lo *et al*. in 2009 [40]. They applied data mining technique to mining the iterative patterns in program traces as the features for supervised learning. More recently, Almaghairbe and Roper conducted a series of experiments on the uses of classifiers of program behaviours as test oracles [1]-[5]. They demonstrated that a reasonably high accuracy can be achieved in this approach.

The approach of learning a classifier can also perform well when it is not easy to give a precise definition of the correctness of a program output and when it depends on fuzzy human evaluation. By using a ML algorithm, a model can be trained with human evaluations of the correctness of program output as the training data. The work by Frounchi et al. in 2011 is a typical example of this type [26].

Frounchi *et al*. are concerned with the test oracle problem in regression testing of programs that produce

complicated output, such as image segmentation in the context of medical imaging [26]. Image segmentation is to identify groups of pixels in a 2D or 3D image that belong to a certain type of objects. It is widely used in the many applications involving image processing, for example, in object measurement or object recognition for diagnosis or treatment planning in medical imaging. The development of an image segmentation algorithm is an iterative process that the program is repeatedly tested on a set of images and revised according to domain experts' feedbacks on the correctness of the output. The complexity of the test oracle problem stems from the feature that even a ground truth correct segmentation of a test case exists, for a clinical task there is typically a range of segmentations that are considered to be correct (e.g., two clinicians often provide slightly different manual segmentations). At the same time, even seemingly small deviations from the ground truth segmentations can have clinical importance. Therefore, the test oracle problem is how to judge two segmentations of the same image are consistent with each other, where one segmentation is an expected output while the other is the output by the program. Frounchi et al. tackled this problem through ML. They trained a ML model to distinguish between consistent and inconsistent segmentation pairs, where the (dis)similarity between different segmentation pairs are quantified using several measures and their consistency is determined from expert evaluations of the first few versions of the segmentation algorithm. Figure 14 shows the activity flow in UML activity diagram for the training and using the test oracle with the evolution of the segmentation software [26]. The swim lane on the right-hand-side labelled as "Learning classifier" is the process that the test oracle is trained. On the left-hand-side, the segmentation algorithm evolves, generates training data and uses the test oracle.

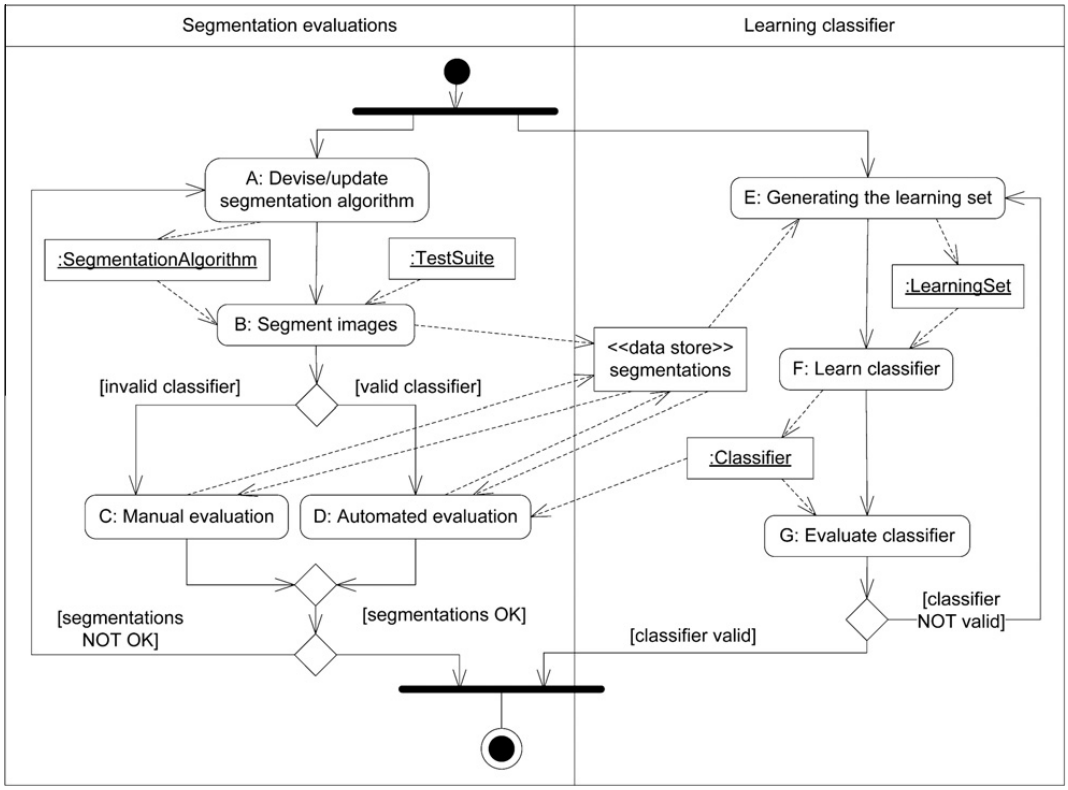


Figure 14. Activity Flow of Training And Using a Test Oracle [26]

The particular ML algorithms used in the case study are J48, JRIP and PART. J48 implements the C4.5 algorithm that creates decision trees. PART uses partial decision trees to construct rules from the branches that lead to a leaf node covering the most instances. JRIP implements the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm, which is a rule-induction technique. For each class, JRIP starts by finding a rule that covers most of the training instances and has the best success rate (least number of misclassified instances). This procedure is repeated recursively until all instances are covered for that class and then repeated for the other classes. The use of decision branches and rules rather than neuron

networks in these ML algorithms allows technical and medical experts to easily interpret the classifiers and gain more confidence in the decisions made by the classifier and the overall approach. Their case study on the proposed approach with 3D segmentations of the cardiac left ventricle obtained from CT scans was successful with promising results with an accuracy of 95%.

Behaviour classifiers use more information to train a model than output predictors, thus they can generate test oracles of better accuracy. However, they also require more effort to label the training data. Classifiers are checkers while predictors are generators. In some cases, checkers are of less computational complexity than the equivalent generators. In such cases, checkers are easier to learn than the corresponding predictor.

It is worth noting that, as studied in the application of ML to the test adequacy problem, a test set is adequate if it can learn a test oracle with sufficiently low error rate and high enough confidence. In other words, if we can train a test oracle with a test set, then the testing is already adequate. There is no need to perform further testing, and thus no need to check the correctness on new test cases. The value of ML in solving the test oracle problem is very limited to certain special situations, such as regression testing.

4.4 Learning Specifications

One of the existing solutions of the test oracle problem is to employ formal specification to check the correctness of the test results [33]. The question is how to get a formal specification. The approach taken by Kanewala and Bieman (2013) is to apply a ML technique to generate metamorphic relations, which can be regarded as a kind of formal specification of software properties [35].

Metamorphic testing was proposed in [15] as a technique for the test oracle problem as well as a test case generation technique. Its basic idea is to use metamorphic relations as the criteria of program correctness.

Definition 4.1. (*Metamorphic Relations*)

Let program p under test be a function on input domain D that produces output in codomain C. Let $K \geq 2$ be a natural number. A K-ary metamorphic relation M is a relation on $D^K \times C^K$ such that program p is correct on input $x_1, x_2, \ldots, x_K$ in D implies that $M(x_1, .., x_K, p(x_1), \ldots, p(x_k))$ holds, where $p(x)$ is program p's output on input x. □

For example, for a program that computes $Sin(x)$ function on real numbers, the following is a metamorphic relation.

$$x_1 + x_2 = \pi \Rightarrow Sin(x_1) = Sin(x_2).$$

Given such a metamorphic relation, to apply metamorphic testing technique, the tester generates test cases x_1 and x_2 that satisfy the condition $x_1 + x_2 = \pi$, executes the program under test on these two test cases, and records the test results $Sin(x_1)$ and $Sin(x_2)$. If these test results are not equal to each other, an error in the software is detected.

Experiments reported in the literature shown that using a set of well designed metamorphic relations can detect a high proportion of bugs in the software; see [49] and [15] for recent surveys of the research on metamorphic testing. It is particularly useful when a complete formal specification of the software is not available. However, when the set of metamorphic relations does not form a complete specification, it only partially ensures correctness. Here, as defined in [70], a test oracle is capable of partially ensuring correctness means that if the program fails a test according to the oracle implies that the program is not correct on the test case. On the other hand, if the program passes a test according to the oracle, it does not guarantee the program is correct on the test case. A key issue with metamorphic testing is to develop a set of metamorphic relations for the software under test. Unfortunately, finding metamorphic relations is a non-trivial task.

As illustrated in Figure 15, Kanewala and Bieman's approach consists of three steps. The first step is to construct a flow graph model from the source code of the program under test. The second step is to extract features from the flow graph and a ML technique is applied to the features to build a predictive model. Finally, metamorphic relations are generated from the predictive model.

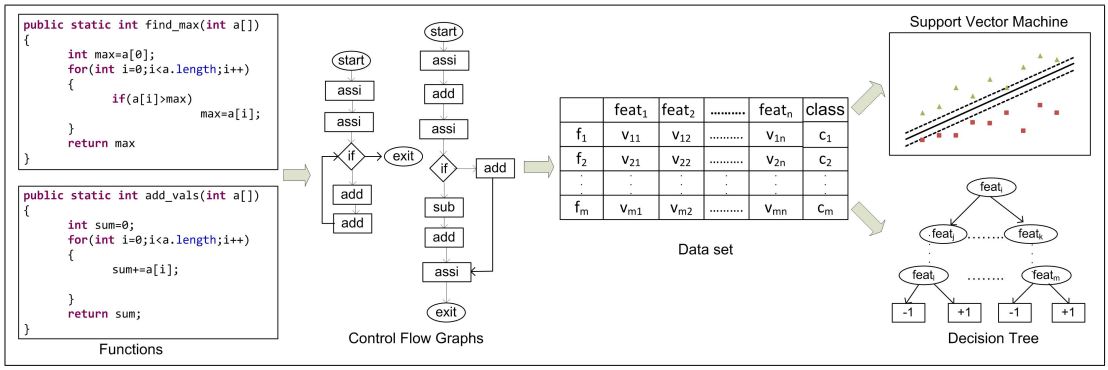


Figure 15. Illustration of Kanewala and Bieman's Method [35]

It is worth noting that, firstly, Kanewala and Bieman were only concerned with a specific type of metamorphic relations called algebraic relations, which are given in Table 4. It is unclear whether the approach can be generalised to other types of metamorphic relations. Secondly, Kanewala and Bieman require the availability of the program's source code. In such cases, more accurate formal specifications of the program can be generated automatically, for example, by using symbolic execution tools. Moreover, the program under test should always satisfy such a formal specification if it is generated from the source code even if it contains bugs. Thus, the test oracle will not detect any fault. Finally, in general, learning a formal specification of the program under test is a kind of explanatory ML. As Theorem 2.1 indicates, explanatory ML is much harder than behavioural ML, which neural networks belong to. Therefore, it is hard to be convinced of any practical value of such a technique.

Table 4. The Type of Metamorphic Relations Studied by Kanewala and Bieman [35]

Relation	Change made to the input
Additive	Add or subtract a constant
Multiplicative	Multiply by a constant
Permutative	Randomly permute the elements
Invertive	Take the inverse of each element
Inclusive	Add a new element
Exclusive	Remove an element
Compositional	Combining two or more inputs

5. Test Case Design

The test case design problem is concerned with how to generate test cases, how to reduce test costs by selecting a subset of existing test cases without significantly compromising the test effectiveness, and how to prioritise test cases so that limited resources can be spent on the most important and effective test cases. It is one of the most intensively studied problems of software testing, but it is still very much an open problem; see [6] for a recent survey on test case generation. ML techniques have been employed to address the test design problem in the following categories.

(a) Supervised learning to predict test case's fault detecting ability so that test cases of high fault detecting abilities are given higher priorities.

(b) Unsupervised learning to clustering test cases and/or the test results so that the similarity between test cases and cluster sizes can be used to guide test case selection.

(c) Reinforcement-like learning to predict the feasibility of a path in the program so that infeasible paths can be eliminated from the test set.

(d) To relate input domain conditions to output properties so that partitioning of the input space can be established for test case generation, and redundant partitions can be removed to reduce the number of test cases.

The following subsections review the work in each of these categories.

5.1 Predicting Test Case's Fault Detection Ability

In 1995, von Mayrhauser, Anderson and Mraz reported their application of neural network to predict the fault detection abilities of test cases in the context of domain-based testing method [56][7]. The training data consists of test cases, the types of faults and severity levels of faults if a test case detects a fault in the software. A neural network model is then constructed so that for newly generated test cases, the severity level of each test case can be predicted. Test cases that are predicted not to detect any fault are removed from test while test cases that are predicted to detect sever faults are selected. Therefore, test effectiveness can be improved.

The experiments reported in the paper employed the domain-based test generation tool Sleuth to generate test data, and used a synthetic test oracle in that it did not model actual software behaviour to evaluate the each test case for error classification. They trained four neural networks using backpropagation. Each network predicts one fault severity level. Each of the neural networks contains 21 input nodes and 1 output node. The number of hidden units was worked out experimentally to achieve the best Root Mean Square (RMS) error during training. The test data set for neural network training included 180 observations, thirty test cases from each of the six test subdomains. In their experiments, the accuracy (i.e. the rate of correct classification) for each level of severities ranges from 82.8% to 94.4%.

5.2 Clustering Test Cases

The basic idea of this category of works is to classify test cases according to certain measures of similarity such that if one or more test cases in a class detect a fault in the software, the other elements of the same class can be predicted to also detect faults. Therefore, the elements in the same class are more effective to use as test cases. Works in this category vary on how test cases are represented and how learning is performed.

As one may expect, test cases that only contain input-output values contain too little information to be meaningfully classified. Successful work reported in the literature all contain more information. In fact, all successful works in this category used profiles of software behaviours. There is a variety of the forms of profiling software behaviours, including statement / basic-block profiling, branch profiling, path profiling, function-call profiling, and various forms of data flow profiling.

Dickinson, Leon, and Podgurski (2001) [20] were concerned with reducing the cost of observation-based software testing, which involves the following steps.

(a) Taking an existing set of program inputs (possibly quite large).

(b) Executing an instrumented version of the software under test on those inputs to produce execution profiles characterizing the executions.

(c) Analysing the resulting profiles, and selecting a subset of the profile for evaluating their conformance to requirements.

Usually, a substantial manual effort is required to evaluate executions profiles for software's conformance to the requirements, or equivalently, to determine the correctness of the software on the test cases. Dickinson, Leon, and Podgurski seek to reduce this effort by filtering out a subset of the original set of executions that is more likely to contain failures than is a randomly chosen subset. They employed agglomerative hierarchical clustering technique to group execution profiles into clusters. Then, one or more execution profiles are selected from each cluster. Moreover, their previous experiment results on clustering test execution profiles suggested that software failures are often isolated in small clusters [46][45]. Thus, a higher priority can be given to small clusters.

Dickinson, Leon, and Podgurski [20] used various dissimilarity metrics to evaluate two strategies of cluster sampling and compared them against the simple random sampling strategy in terms of their effectiveness for locating the failures present in the populations of profiles. *One-per-cluster* sampling involves selecting one execution at random from each cluster. *Adaptive* sampling involves initially selecting one execution at random from each cluster and then including the remaining executions from the cluster if the first one selected from it is a failure. The main results of their experiments are: (1) cluster filtering is more effective than simple random sampling for finding failures in populations of operational executions; (2) adaptive

sampling is more effective than one-per-cluster sampling; and (3) dissimilarity metrics which give extra weight to unusual profile features are most effective.

Bowring, Rehg and Harrold (2004) improved the software behaviour clustering technique by an active-learning paradigm to replace batch-learning [12]. In active learning, the classifier is trained incrementally on a series of labelled data elements. They also explored the use of Markov processes as models of program executions. These Markov models of individual program executions can be clustered and aggregated into effective predictors of program behaviour.

Yoo, Harman, Tonella and Susi (2009) [62] also used agglomerative hierarchical clustering to classify test cases according to their execution paths. However, the result of clustering was used to prioritise the classes of test cases by manual comparisons between the classes of test cases. One of the features of agglomerative hierarchical clustering is that it is possible to generate an arbitrary number of clusters. This allows for control of the number of comparisons presented to the human tester. Therefore, the number of required comparisons by domain expert can be significantly reduced. Their empirical evaluation of the technique showed that such class prioritisations are more effective than pairwise prioritisation technique in terms of fault detecting ability.

5.3 Predicting Path Feasibility

In structural testing, a test case corresponds to a complete path in the program from the entry node to the exit node. However, a complete path may be infeasible. That is, there is no input data on which the program can execute through the path. In general, it is not decidable whether a path is feasible, thus it often relies on manual check to decide whether a path is feasible and test data for the path can be worked out. How to check automatically the feasibility of a program path is a long lasting research problem in structural testing.

Baskiotis *et al*. addressed this problem through a ML approach [11]. After failed to solve the problem in a discriminant learning approach, Baskiotis *et al*. [11] proposed a new ML algorithm called EXIST, which stands for Exploration - eXploitation Inference for Software Testing. EXIST employs a probability distribution on program paths to generate a path with highest probability of being feasible. This probability distribution is updated after each time a new path is generated and labelled manually as feasible or infeasible. It proceeds iteratively through cycles of generating a path and updating the distribution according to the label assigned to the path. In this sense it is similarity to reinforcement learning, but its goal is neither to learn a concept nor a fixed policy. Instead, it aims at maximising the number of distinct feasible paths found along the process of generating program paths. It is also worth noting that Baskiotis *et al*. extended Parikh map by providing a powerful propositional description of long structured sequences to represent program paths. Their experimental data show that the proposed approach can dramatically increase the ratio of (distinct) feasible paths generated, compared to the uniform sampling.

5.4 Relating Input-Output Features

In section 4, we have seen work by Last and Kandel and their colleagues on application of ML techniques to test oracle problem. Their group also made a contribution to using ML for the test design problem [48].

Saraph, Last and Kandel are concerned with reducing the number of test cases in black-box testing [48]. In particular, they are concerned with the situation when the software under testing has a large number of input variables, and the values of each input variable that can be divided into a relatively small number of subsets. Even if a small number of test data in each subset is adequate to test the software, the number of combinations of these data over all input variable can be huge. However, some of the combinations are ineffective and unnecessary to be tested on, for example, because the computation does not use values of two variables at the same time. Analysis of the input-output relationship can help to reduce the number of test cases by removing unnecessary combinations of input data. Saraph, Last and Kandel's approach consists of four steps as shown in Figure 16.

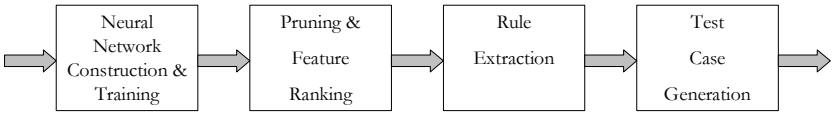


Figure 16. Saraph, Last and Kandel's Process of Test Case Generation and Reduction

At the first step, back-propagation is employed to train a neural network about the relationships between input-output. The training data are the features of input-output pairs. The second step is to prune the neural network to the minimal size and to rank the features according to the weights associated to the links between neurons. The third step is to extract rules of the relationships between input and output from trained and pruned neural network. These rules can then be used in step four to generate test cases and also to reduce the number of test cases in an existing test set. Saraph *et al*. demonstrated the process with the credit card approval software, which was also used in their earlier work on the application of ML to test oracle problem [55].

6. Testability

Testability is a quality attribute of computer software. The testability problem is concerned with how to define the notion of testability, how to measure the testability of a software under test, and how to improve testability of a software system, for example, by transformation of the code while preserving the functional correctness. However, the application of ML to testability problem so far is limited to understand the notion of testability.

In comparison with learnability, testability is a much less investigated subject area. Freedman defined the notion of software testability as the conjunction of observability and controllability of software [24]. It is probably the most widely referenced definition of testability in the literature. In this definition, observability refers to the ease of determining if specified inputs affect the outputs. Controllability refers to the ease of producing a specified output from a specified input. Another popular definition of testability is given by Bache and Mullerburg [9]. They defined testability in the context of a test method, or in fact, with respect to an adequacy criterion. That is, the testability of a program is measured by the minimum number of test cases to provide total test coverage, assuming that such coverage is possible.

In Gourlay's mathematical framework of software testing [29], a program p is testable with respect to a specification s if there exists a finite test set such that the program is correct with respect to s if and only if p is correct on all of the data in the test set. Such a test set is called a *reliable* test set, following a notion from Goodenough and Gerhart [28]. Therefore, the power (or scope) of a testing method can be defined as the set of (reliably) testable functions with respect to a set of corresponding specifications. Gourlay proved some ordering of sets of functions on the power of testing methods. This approach is very similar to the study of the power of inductive inference methods, for example, the results shown in Figure 1.

The link between testability and learnability was explored by Budd and Angluin [13], where the notion of testability is defined based on distinguishability, which is a fundamental concept underlying mutation testing.

Definition 6.1 (Testability as Distinguishability)

A set P of programs on a domain D is said to be *testable*, if there is a mapping T from P to finite subsets of D such that for all programs p and q in P there is at least one data d in $T(p)$ such that $p(d) \neq q(d)$. □

Budd and Angluin compared this notion of testability with the notion of learnability based on logic identification [13]. Cherniavsky and Smith [16] used the explanatory learning with identification in the limit to compare testability and learnability. Cherniavsky and Statman [18] further proposed the notions of *fixed time testability*, *finite time testability*, and *testability in the limit*.

Definition 6.2 (Test in The Limit)

Let P be a set of functions. The set P of functions is *fixed time testable* if the set of functions are distinguishable from each other using a fix number of test cases that the number is independent of the function to be distinguished. The set P of functions is *finite time testable* if the set of functions is distinguishable using a number of test cases that the number may depend upon the function to be distinguished, but independent of any ordering on the set of functions. The set P of functions is *testable in the limit*, if there is a well ordering of $P = \{p_1, p_2, \dots, p_n, \dots\}$, such that there exists a function T that maps each p_i in P to a finite test set $T_i = T(p_i)$ with the property that for all $j < i$, there is some x in $T(p_i)$ such that $p_i(x) \neq p_j(x)$. □

Cherniavsky and Statman's definition of finite time testable is exactly what Budd and Angluin's notion of testable is. An example of fixed time testable set of functions is the set of one-variable linear functions,

which can be distinguished one from another by two different data. A set of functions is *testable in the limit*, if the number of test cases may be dependent upon both the function to be distinguished and some underlying ordering of the set of functions. An example of sets of functions that are testable in the limit is the set of polynomial functions. A unary polynomial function of degree k can be distinguished from all the polynomial functions whose degrees are less than or equal to k, by a test set of $k + 1$ data. Testable in the limit means that it may be impossible to distinguish a function using a finite test set. Hence, testing such a function needs an infinite sequence of test data. And, at any finite stage of testing, it is not known whether testing has already been successful. In this sense, testing in the limit is a counterpart of identification in the limit.

Instead of requiring the set of programs being well ordered, Davis and Weyuker [19] considered the situation when the set of programs P to be distinguished from a program p is determined by a metric (or distance function) on the program space. That is, P is the set of programs within a given distance d from the program p according to the metric.

Definition 6.3 (David-Weyuker Testable)

Let P be a set of programs and $\delta(p, q)$ be a distance metric defined on P. The set of programs P is David-Weyuker testable with respect to $\delta(p, q)$, if for all programs p in P and every given number $d > 0$, there is a finite test set T such that

$$\forall q \in P. \Big(\delta(p, q) < d \wedge (p \neq q) \Rightarrow \exists t \in T. \big(p(t) \neq q(t)\big)\Big).$$

□

It is easy to see that if a set of functions $P = \{p_1, p_2, \dots, p_n, \dots\}$ is testable in the limit, then it is David-Weyuker testable with respect to the distance function $\delta(p_i, p_j) = |i - j|$. Therefore, David-Weyuker testability is a further generalization of testability in the limit.

On the other hand, let P be a set of functions that is David-Weyuker testable with respect to a distance metrics $\delta(p, q)$ on . If the distance function has the property that there is a function p_0 in P such that

$$\forall d < \infty. (\|\{p \in P | \delta(p_0, p) < d\| < \infty),$$

$$\forall p \in P. \big(\exists d < \infty. (\delta(p_0, p) < d)\big),$$

then, it is also testable in the limit. Note that, if the distance metrics has the above properties, the functions in the set P can be totally ordered according to the distance to p_0.

However, David-Weyuker testable does not imply that the correctness of the program can be proved for through successful testing. For example, let P be all the programs written in a particular programming language, such as Java. For any given programs p and q in P, we define the *mutation distance* between p and q as $\delta(p, q) = k$, if q is a k'th order mutant of p. Therefore, the set of all programs in a programming language is David-Weyuker testable with respect to the mutation distance. However, successfully testing a program on a test set that can distinguish the program from all non-equivalent mutants of a given order k does not imply that the program is correct. This example shows that the testable by distinguishability does not implies the program's correctness with respect to a specification. A similar example can be constructed for testable in the limit because the number of mutants of any given order is finite.

The main conclusions that we can draw are two folds. First, a function is learnable implies that it is testable. Thus, learning is a more difficult computational problem than testing. Second, when a ML technique is used to define testability, its inductive inference power (i.e. the set of functions that is learnable for the inference device) determines the set of functions that are testable.

7. Conclusion

In this paper, we reviewed the current state of the research on application of ML techniques and theories to solve software testing problems as reported in the literature. Existing work can be classified into four categories according to the problems addressed while some research efforts have developed techniques that

can be applied to address multiple problems of software testing. The problems that have been tackled are (1) test adequacy problem, i.e. how to measure test adequacy, (2) test oracle problem, i.e. how to check the correctness of the outputs and execution behaviours of software on test cases; (3) test case design problems, how to generate test cases, prioritize test cases, and reduced the sizes of test suites, and (4) testability problem, i.e. how to measure software's testability. These are among the most important but difficulty problems of software testing. For each of these problems, various approaches employing ML are identified summarised, and analysed.

The following general observations on the current state can be made from the review. For each of them, the research directions to break through the current state if possible are discussed below.

(1) *No easy money*. Applying ML techniques to software testing is labour intensive and expensive. It is not economically viable as a practical technique, yet.

In 2008, after a critical review of his own research on the application of ML to software testing, Brian [38] pointed out that "*There is very little evidence, for all existing applications of ML in software testing, that such [ML] techniques bring any benefits.*" The situation has not changed in the past ten years since then. Generally speaking, applying ML technique to solve a software testing problem is a labour intensive, difficult and frustrated task. The literature has not provided any good evidence that the cost of applying ML techniques to software testing could be reduce by using automated or semi-automated tools. This is because ML requires a large number of data to train the model. For supervised learning, this means a large amount of data to be labelled, which are mostly done manually. Most of the successful research works reported in the literature use unsupervised learning techniques like clustering, such as the works by Podgurski *et al*. [20]. Theoretically speaking, the literature on the study of the relationships between testability and learnability shows that learning is computationally harder then testing. A hypothesis is that applying ML to solve testing problems is inevitably costly. However, one situation has been identified that ML is potentially beneficial for software testing, that is, regression testing in software evolution process where the cost of ML can be reduced by reusing the training data and/or the train results. A question worth further research is on how to enable ML algorithms adapting with the evolution of the learning target so that it can be applied to testing problems in the context of software evolution.

(2) *No free lunch*. Wolpert's *No-Free-Lunch* theorem is observed in the experiments with the application of ML to software testing problems. That is, there is no single ML algorithm that consistently performed better than other ML techniques.

Fraser and Walkinshaw pointed out that in their experiments different ML techniques performed differently on different programs under test. Moreover, it is impossible for them to predict which technique will perform better than the others for a given software system [23]. This phenomenon is a mathematical folklore called *No Free Lunch Theorem*, which was proved by Wolpert for ML in 1996 [60]. A direction for future research is how to express the context of the software under test in order to apply the PAC ML theory and select an appropriate ML algorithm. For example, to generate a test oracle from test cases, can the software under test be considered as an element of a set of learnable rules? And how can we derive the VC-dimension of the set of rules?

(3) *No silver bullet*. Existing ML techniques cannot deal with the complexity confronted software engineers in current practices.

A variety of ML techniques have been explored to solve software testing problems. None of these techniques are powerful enough to provide convincing results to show ML as a promising solution to software testing problems. In particular, none of the ML techniques are capable of dealing with the complexity of software systems. Especially, none of them can process program codes, software models, and formal specifications etc. as ML algorithms' input. Although there are many advanced ML techniques (such as recurrent neural networks) that have not been employed in the research, it is fair to claim that existing ML techniques are not powerful enough to process such structurally complex data. A direction for future research is to study how to learn from structurally complex data.

Although applications of ML to software testing problems are still far away from practical uses, research on this topic has shed a new light to software testing problems. There are potentially profound and significant impacts that possibly foster a paradigm shift of software testing practice. In particular, a new conceptual model of software testing can be built based on regarding software testing as inductive inference as

discussed in Section 3. In this new model, an assertion on software reliability could be a probability statement, e.g. the δ-probable reliability of software is ε. The assessment of software reliability by testing could be guided by the complexity or testability of the software under test, where testability and complexity could be measured by VC-dimension or something similar. This could significant change the practice of software test engineering by including complexity/testability analysis as a part of testing process. It will enable software testing be based on a solid theoretical foundation.

REFERENCES

[1] Agarwal, D., Tamir, D. E., Last, M. and Kandel, A., A Comparative Study of Artificial Neural Networks and Info-Fuzzy Networks as Automated Oracles in Software Testing, IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, vol.42, no.5, pp. 1183-1193, Sept. 2012.

[2] Almaghairbe, R. and Roper, M., Automatically Classifying Test Results by Semi-Supervised Learning, Proc. of IEEE 27th International Symposium on Software Reliability Engineering (ISSRE 2016), pp. 116-126, 2016, ISSN 2332-6549.

[3] Almaghairbe, R. and Roper, M., Building Test Oracles by Clustering Failures, Proc. of IEEE/ACM 10th International Workshop on Automation of Software Test (AST 2015), pp. 3-7, 2015.

[4] Almaghairbe, R. and Roper, M., Separating passing and failing test executions by clustering anomalies, Software Quality Journal, pp.803-840, 2017(25), ISSN 0963-9314.

[5] Almaghairbe, R., Formulating Test Oracles via Anomaly Detection Techniques. PhD Thesis, University of Strathclyde, UK, 2017.

[6] Anand, S., *et al*., An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software, Vol. 86, No. 8, pp1978-2001, Aug. 2013.

[7] Anderson C, von Mayrhauser A, Mraz R. On the use of neural networks to guide software testing activities. In Proceedings of the International Test Conference (ITC'95), October 21–26, 1995.

[8] Angluin, D., Inductive Inference of Formal Languages from Positive Data, Information and Control 45, pp117–135, 1980. (doi:10.1016/S0019-9958(80)90285-5).

[9] Bache, R. and Mullerburg, M., Measures of testability as a basis for quality assurance, Software Engineering Journal, 5 (2), pp86-92, March 1990.

[10] Baker, A. L., Howatt, J. W., and Bieman, J. M., Criteria for finite sets of paths that characterize control flow. In Proceedings of the 19th Annual Hawaii International Conference on System Sciences, pp158–163, 1986.

[11] Baskiotis, N., Sebag, M., Gaudel, M.-C. and Gouraud, S., A Machine Learning Approach for Statistical Software Testing, Proc. International Joint Conference on Artificial Intelligence, 2007.

[12] Bowring J. F., Rehg J. M. and Harrold M. J., Active Learning for Automatic Classification of Software Behavior, Proc. ACM International Symposium on Software Testing and Analysis, 2004.

[13] Budd, T. A. and Angluin, D., Two notions of correctness and their relation to testing, Acta Informatica, 18, pp31-45, 1982.

[14] Case, J. and Smith, C., Comparison of identification criteria for machine inductive inference, Theoretical Computer Science**,** 25(2), 193-220, 1983.

[15] Chen, T.Y., et al., Metamorphic Testing: A New Approach for Generating Next Test Cases, Technical Report HKsUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[16] Chen, T. Y., Kuo, F-C., Liu, H., Poon, P-L., Towey, D., Tse, T. H., Zhou, Z. Q., Metamorphic testing: a review of challenges and opportunities. ACM Computing Surveys, 51 (1). 4/1-4/27, 2018.

[17] Cherniavsky, J. C. and Smith, C. H., A recursion theoretic approach to program testing, IEEE Transactions on Software Engineering, 13 (7), pp777-784, 1987.

[18] Cherniavsky, J. C. and Statman, R., Testing: an abstract approach, Proc. of the Second Workshop on Software Testing, Verification and Analysis, Banff, Canada, pp38-44, July 1988.

[19] Davis, M. and Weyuker, E., Metric space-based test-data adequacy criteria, The Computer Journal, 13 (l), pp17-24, February 1988.

[20] Dickinson, W., Leon, D., Podgurski, A., "Finding failures by cluster analysis of execution profiles", Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 339-348, May 2001.

[21] Eibe Frank, Mark A. Hall, and Ian H. Witten, Data Mining: Practical Machine Learning Tools and

Techniques, Morgan Kaufmann, Fourth Edition, 2016.
[22] Fraser, G. and Walkinshaw, N., Assessing and generating test sets in terms of behavioural adequacy. Software Testing, Verification And Reliability 25 pp749–780, 2015.
[23] Fraser, G., and Walkinshaw, N., Behaviourally adequate software testing. In Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST 2012), Antoniol, G., Bertolino, A., Labiche, Y. (eds), Montreal, QC, Canada, pp300–309, April 17–21, 2012.
[24] Freedman, R. S., Testability of software components, IEEE Transactions on Software Engineering, 17 (6), pp555-564, 1991.
[25] Freund, Y. and Schapire R., A decision-theoretic generalization of on-line learning and an application to boosting. In Computational Learning Theory. Springer: Jerusalem, Israel, 1995; 23–37.
[26] Frounchi, K., Briand, L. C., Grady, L., Labiche, Y. and Subramanyan, R., Automating image segmentation verification and validation by learning test oracles, Inf. Softw. Technol., vol. 53, no. 12, pp. 1337-1348, Dec. 2011.
[27] Gold, E. M., Language identification in the limit, Information and Control 10, pp447-474, 1967.
[28] Goodenough, J. B. and Gerhart, S. L., Toward a theory of test data selection, IEEE Transactions on Software Engineering, 1 (2), pp156-173, June 1975.
[29] Gourlay, J., A mathematical framework for the investigation of testing, IEEE Transactions on Software Engineering, 9 (6), pp686-709, 1983.
[30] Hahnloser, R , Sarpeshkar, R., Mahowald, M A , Douglas, R. J. and Seung, H.S., Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature. 405. pp. 947–951, 2000
[31] Hamlet, R., Theoretical comparison of testing methods. In Proceedings of The 3rd SIGSOFT Symposium on Software Testing, Analysis, and Verification, pp28–37, Dec. 1989.
[32] Hamlet, R., Probable correctness theory, Information Processing Letters, 25(1), 17-25, 1987.
[33] Hierons, R. M., et al., Using formal specifications to support testing. ACM Comput. Surv. 41, 2, Article 9 (February 2009), 76 pages. DOI: https://doi.org/10.1145/1459352.1459354
[34] Hutchinson, A., Algorithmic Learning, Graduate Texts in Computer Science, Oxford University Press, 1994.
[35] Kanewala, U. and Bieman, J. M., Using machine learning techniques to detect metamorphic relations for programs without test oracles, Proceedings of IEEE 24th International Symposium on Software Reliability Engineering (ISSRE 2013), pp. 1-10, 2013.
[36] Kubat,M., An Introduction to Machine Learning, 2nd Edition, Springer, 2017.
[37] Last, M. and Maimon, O., A compact and accurate model for classification, IEEE Trans. Knowl. Data Eng., vol. 16, no. 2, pp. 203–215, Feb. 2004.
[38] Lionel C. Briand, Novel Applications of Machine Learning in Software Testing, The Eighth International Conference on Quality Software, 2008, pp3-10.
[39] Liu, L. and Miao, H., Axiomatic Assessment of Logic Coverage Software Testing Criteria, Journal of Software 15(9), pp1301-1310, September 2004. (In Chinese)
[40] Lo, D., Cheng, H., Han, J., Khoo, S.-C., Sun, C., Classification of software behaviors for failure detection: A discriminative pattern mining approach, Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 557-566, 2009.
[41] Menon, A. K., Tamuz, O., Gulwani, S., Lampson, B. and Kalai, A. T., A Machine Learning Framework for Programming by Example, in Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28, ppI-187-I-195.
[42] Parrish, A. and Zweben, S. H., Analysis and refinement of software test data adequacy properties. IEEE Trans. Softw. Eng. SE-17(6), pp565–581, Jun. 1991.
[43] Parrish, A. and Zweben, S. H., Clarifying some fundamental concepts in software testing. IEEE Trans. Softw. Eng. 19(7), pp742–746, Jul. 1993.
[44] Podgurski A., Leon D., Francis P., Masri W. and Minch M., Automated Support for Classifying Software Failure Reports, in Proc. of the 25th International Conference on Software Engineering (ICSE 2003), 2003.
[45] Podgurski, A. and Yang, C. Partition testing, stratified sampling, and cluster analysis. Proceedings of the First ACM Symposium on Foundations of Software Engineering (Los Angeles, CA, December 1993), ACM Press, 169-181.
[46] Podgurski, A., Masri, W., McCleese, Y., Wolff, F.G., and Yang, C. Estimation of software reliability by stratified sampling. ACM Transactions on Software Engineering and Methodology 8, 9 (July,

1999), 263-283.
[47] Quinlan J. R., C4. 5: Programs for Machine Learning. Morgan Kaufmann: San Mateo, CA, 1993.
[48] Saraph, P., Last, M. and Kandel, A., "Test set generation and reduction with artificial neural networks in Artificial Intelligence Methods in Software Testing, Singapore: World Scientific, pp. 101-132, 2004.
[49] Segura, S., Fraser, G., Sanchez, A. B. and Ruiz-Cortés, A., A Survey on Metamorphic Testing, in IEEE Transactions on Software Engineering, vol. 42, no. 9, pp. 805-824, 1 Sept. 2016. doi: 10.1109/TSE.2016.2532875
[50] Shahamiri, S., Wan-Kadir, W., Ibrahim, S. and Hashim, S., Artificial neural networks as multi-networks automated test oracle", Automated Software Engineering, vol. 19, no. 3, pp. 303-334, 2012.
[51] Shalev-Shwartz, S. and Ben-David, S., Understanding Machine Learning: From Theory to Algorithms, Cambridge University Press, 2014.
[52] Smith, C. H. and Angluin, D., Inductive inference: theory and methods, ACM Computing Surveys, 15(3), 235-269, 1983.
[53] Summers, P. D., A methodology for LISP program construction from examples. Journal of ACM 24(1), pp161-75, Jan. 1977.
[54] Valiant, L. C., A theory of the learnable, Communications of the ACM, 27(11), pp1134-1142, 1984.
[55] Vanmali, M. , Last, M. and Kandel, A., Using a neural network in the software testing process, International Journal of Intelligent Systems, vol. 17, no. 1, pp. 45-62, 2002.
[56] von Mayrhauser, A., Anderson, Ch. and Mraz, R., Using A Neural Network to Predict Test Case Effectiveness, in Proceedings of IEEE Aerospace Applications Conference, Snowmass, CO, Feb. 1995.
[57] Weyuker, E. J. The evaluation of program-based software test data adequacy criteria, Communications of the ACM, 31(6), pp668-675, 1988.
[58] Weyuker, E. J., Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems, 5(4), pp641-655, 1983.
[59] Weyuker, E. J., Axiomatizing software test data adequacy, IEEE Transactions on Software Engineering, 12(12), pp1128-1138, 1986.
[60] Wolpert D.H., The lack of a priori distinctions between learning algorithms. Neural Computation 1996; **8**(7), pp1341–1390.
[61] Xavier Glorot, Antoine Bordes and Yoshua Bengio(2011). Deep sparse rectifier neural networks (PDF). AISTATS.
[62] Yoo, S., Harman, M., Tonella, P. and Susi, A., Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge, Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2009), pp. 201-211, July 2009.
[63] Zhu, H. and Hall, P. A. V. Test data adequacy measurements, Softw. Eng. J. 8(1), pp21–30, Jan. 1993.
[64] Zhu, H. and Jin, L., A knowledge-based approach to program synthesis from examples, Journal of Computer Science and Technology, January 1991.
[65] Zhu, H. and Jin, L., A knowledge-based system to synthesize FP programs from examples, Proc. of EPIA'89, Lecture Notes in Computer Science, Vol. 390, 1989.
[66] Zhu, H., A formal interpretation of software testing as inductive inference. Journal of Software Testing, Verification and Reliability 6, pp3-31, 1996
[67] Zhu, H., Axiomatic assessment of control flow based software test adequacy criteria. Softw. Eng. J., pp194–204, Sept. 1995.
[68] Zhu, H., Hall, P. and May, J., Inductive inference and software testing. Journal of Software Testing, Verification, and Reliability 2, pp69-81, 1992.
[69] Zhu, H., Hall, P. and May, J., Software unit test coverage and adequacy, ACM Computing Survey, 29(4), pp366~427, Dec. 1997.
[70] Zhu, H., JFuzz: A Tool for Automated Java Unit Testing based on Data Mutation and Metamorphic Testing Methods, Proc. of the 2nd International Conference on Trustworthy Systems and Their Applications (TSA 2015), 8-9 July 2015, Hualien, Taiwan, pp8-15.
[71] Zhu, H., Towards a relationship between software reliability estimation and complexity analysis, Chinese Journal of Software, 9(9), 713-717, Sept. 1998. (In Chinese)
[72] Zweben, S. H. and Gourlay, J. S., On the adequacy of Weyuker's test data adequacy axioms. IEEE Trans. Softw. Eng. SE-15(4), pp496–501, Apr. 1989.