The Journal of Systems & Software 187 (2022) 111231

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Discovering boundary values of feature-based machine learning classifiers through exploratory datamorphic testing[☆],[☆☆]

Hong Zhu[*], Ian Bayley

School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, United Kingdom

A R T I C L E I N F O

Article history:
Received 22 April 2021
Received in revised form 12 December 2021
Accepted 20 January 2022
Available online 31 January 2022

Keywords:
Artificial intelligence
Software testing
Automation of software test
Datamorphic testing
Exploratory testing
Test strategies

A B S T R A C T

Testing has been widely recognised as difficult for AI applications. This paper proposes a set of testing strategies for testing machine learning applications in the framework of the datamorphism testing methodology. In these strategies, testing aims at exploring the data space of a classification or clustering application to discover the boundaries between classes that the machine learning application defines. This enables the tester to understand precisely the behaviour and function of the software under test. In the paper, three variants of exploratory strategies are presented with the algorithms implemented in the automated datamorphic testing tool Morphy. The correctness of these algorithms are formally proved. Their capability and cost of discovering borders between classes are evaluated via a set of controlled experiments with manually designed subjects and a set of case studies with real machine learning models.

1. Introduction

It is widely recognised that the generation of test data for AI applications is prohibitively expensive (Tian et al., 2018). Checking the correctness of a test result is also notoriously difficult, if not completely impossible (Segura et al., 2018; Zhou and Sun, 2019). Moreover, existing testing techniques for measuring test coverage and the automation of testing activities and processes are not directly applicable (Zhu et al., 2018). Testing AI applications is therefore a grave challenge for software engineering (Bai et al., 2018). Developing novel approaches to test AI applications is highly desirable (Gotlieb et al., 2019).

In Zhu et al. (2018, 2019b), we proposed a method called *datamorphic testing* for testing AI applications and reported a case study with AI applications. In Zhu et al. (2019a, 2020) we developed this method further, defined the notion of test morphisms and reported an automated testing tool called *Morphy*. In Zhu et al. (2020), we defined formally a set of test strategies that combine datamorphisms to cover various scenarios in AI applications; Zhu et al. (2019a) reports case studies that show the strategies significantly improve automated in testing AI applications.

In Zhu and Bayley (2020), we proposed another set of strategies to test the classification and clustering variety of AI applications, as they are very common and arise from machine learning and data analytics techniques; see, for example, Aggarwal (2015), Mohri et al. (2012) and Shalev-Shwartz and Ben-David (2014). These strategies are based on the idea of exploratory testing, in which outputs from the previous tests is used to change the focus of testing so that as much as possible of the application's functionality is explored (Whittaker, 2009). Whereas confirmatory testing verifies and validates the correctness of the software under test with respect to a given specification, exploratory testing treats it as an object unknown and conducts experiments to discover its functions and features. The two approaches also differ in their treatment of test cases. Confirmatory testing treats test cases as being mutually independent whereas exploratory testing uses the results of earlier test cases to guide the selection of subsequent test cases. In particular, the strategies in Zhu and Bayley (2020) aim at discovering the borders between classes of a classifier. The main contributions of Zhu and Bayley (2020) are:

- The notion of *Pareto front* was introduced and formally defined to represent borders between classes.
- Strategies to produce Pareto fronts from machine learning models were formally defined as datamorphic testing algorithms.
- The algorithms were formally proved correct and implemented in the Morphy tool.
- Their cost efficiency was demonstrated by conducting controlled experiments with 10 manually coded classifiers as subjects.

[☆] Editor: Raffaela Mirandola.
[☆☆] This paper is an extended and revised version of the conference paper by Zhu and Bayley (2020).
[*] Corresponding author.

E-mail addresses: hzhu@brookes.ac.uk (H. Zhu), ibayley@brookes.ac.uk (I. Bayley).

https://doi.org/10.1016/j.jss.2022.111231
0164-1212/© 2022 Elsevier Inc. All rights reserved.

This paper extends that work and has the following main contributions:

- The notion of completeness is formally defined for a datamorphic test system to be used for exploratory testing.
- A systematic method is proposed for constructing exploratory test systems for any feature-based classifier, which are among the most common types of machine learning applications; their completeness was also proven.
- We extend the evaluation in Zhu and Bayley (2020) by building 48 real machine learning models constructed from 3 real datasets using 8 different machine learning algorithms, in addition to 10 manually coded classifiers already used in Zhu and Bayley (2020). For each strategy, we measure both its cost and its capability of discovering classifier borders. The evaluation found that cost-effectiveness is high for both.

The paper is organised as follows. Section 2 defines the basic concepts underlying the work: the basic notions and notations of machine learning classifiers, the exploratory testing approach, the datamorphic testing method and the automated testing tool Morphy. Section 3 is a theoretical study of the exploratory test systems for various types of feature-based classifiers, which proves that such test systems exist for all such types of feature-based classifiers. Section 4 defines the exploration strategies and illustrates their uses with an example. Section 5 reports the controlled experiments with the 10 manually coded classifiers and 48 machine learning models. Section 6 compares the proposed testing method with related work. Section 7 concludes the paper with a discussion of future work.

2. Preliminaries

In this section, we briefly review the notions and notations underlying our proposed approach.

2.1. Classification applications

Clustering as a data mining and machine learning problem is the partitioning of a given set of data points into groups containing similar data points. The grouping is based on a notion of similarity between data points, defined formally with a distance function on the data space. Two pieces of data that are similar to each other should be put into the same group, whilst data that are dissimilar should be placed in different groups. Whereas clustering is unsupervised learning, classification is supervised learning. Given a number of examples of data points and their classifications, the algorithm learns how to assign data to groups (Aggarwal, 2015; Mohri et al., 2012; Shalev-Shwartz and Ben-David, 2014).

In both clustering and classification, the result is a program P that maps from the data space D into a number of non-empty groups G such that $D = \bigcup_{g\in G}(g)$ and $\forall g, q \in G.(g \neq q \Rightarrow g \cap q = \emptyset)$. We say that P is a *classification application*. We will write $P(x)$ to denote the output of P on an input $x \in D$, and call $P(x)$ the classification of x by P. We also assume that there is a function $\|\cdot,\cdot\| : D \times D \to \mathbb{R}^+$ ($\mathbb{R}^+ = \{x \in \mathbb{R} \mid x \geq 0\}$) measuring the distances between any two points x and y in the data space D, with shorter distance denoting greater similarity, such that:

- $\forall x \in D.(\|x, x\| = 0)$;
- $\forall x, y \in D.(\|x, y\| \geq 0)$;
- $\forall x, y \in D.(\|x, y\| = \|y, x\|)$;
- $\forall x, y, z \in D.(\|x, y\| + \|y, z\| \geq \|x, z\|)$.

For a classification program, it is crucial that data is assigned to the correct classes. However, the borders between classes are often unknown if the classification program is obtained through machine learning and data mining. The goal of the exploratory testing proposed in this paper is to find a set of data pairs that represents the borders between classes. Thus, we introduce the notion of a *Pareto front* for the classification as defined by the program P under test.

Definition 1 (*Pareto Front of Classification*)**.** Let $P : D \to G$ be a classification program, $\|\cdot,\cdot\| : D \times D \to \mathbb{R}^+$ be a distance metric defined on the input space D, and $\delta > 0$ be any given real number. A set $\{\langle a_i, b_i\rangle \mid a_i, b_i \in D, i = 1, \dots, n\}$ of data pairs is a *Pareto front* of the classes of D according to P with respect to $\|\cdot,\cdot\|$ and δ, if for all $i = 1, \dots, n$, $P(a_i) \neq P(b_i)$ and $\|a_i, b_i\| \leq \delta$. □

A Pareto front can show accurately the borders between classes within a tolerable error margin δ. In this way, it helps testers to determine whether the classification is correct or not.

The structure of the data space D determines the type of the classification system. We now define a few standard types that are often seen in the literature.

Definition 2 (*Feature-Based Classifier*)**.** Let $P : D \to G$ be a classification program. We say that P is a *feature-based classifier* if there is a natural number $K \geq 1$ such that $D = D_1 \times \cdots D_K$, where for every $i = 1, \dots, K$, D_i is the set of values of a feature f_i. Moreover, a feature f_i is *discrete non-numerical* if D_i is a finite non-empty set. A feature f_i is *discrete numerical*, if D_i is the set of integer values or natural numbers. A feature f_i is *continuous numerical*, if D_i is the set of real numbers, or a non-empty interval of real numbers. □

As these are disjoint alternatives, a feature-based classifier can further be classified disjointly according to the types of its features.

Definition 3 (*Types of Feature-Based Classifiers*)**.** Given a feature-based classifier $P : D_1 \times \cdots D_K \to G$, where D_i is the domain of feature f_i, we say that

- P is a *discrete non-numerical feature-based classifier* or simply a *discrete non-numerical classifier*, if all features f_i are discrete non-numeric.
- P is a *discrete numerical feature-based classifier* or simply a *discrete numerical classifier*, if all features f_i are discrete numeric.
- P is a *continuous numerical feature-based classifier*, or simply a *continuous numerical classifier* if all features f_i are continuous numeric.
- P is a *hybrid feature-based classifier* or simply *hybrid classifier*, if its data space contains more than one type of features. □

Feature-based classifiers are the most common kind of data analytic and machine learning applications. There are other more complicated classifiers, such as time series classifiers, but in this paper we will only study feature-based classifiers.

Example 1. Consider a classifier that classifies the points in a two-dimensional continuous space $[0, 2\pi] \times [-1, 1]$ into three classes: *red*, *black* and *blue* as illustrated in Fig. 1. This example is a continuous numerical classifier. In this example, data points x and y are a Pareto front pair between *black* and *red* classes, if x is *red* and y is *black* and they are very close to each other. Such pairs can show accurately the borders between classes, and thus help testers to determine whether the classification is correct or not. □

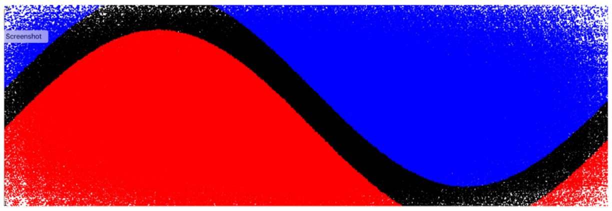


Fig. 1. Data space of the running example. The readers are referred to the web version of this article for a coloured version of this figure.

In the rest of this paper, we will use the above classifier as a running example to explain the definitions of notions and to illustrate the exploration strategies.

2.2. Exploratory testing

Although exploratory testing (ET) has been widely practised in the industry for a long time, the first use of the term "exploratory testing" was in a book by Kaner (1988). It takes a pragmatic approach to software testing under normal business conditions and is based on his experiences as a software testing engineer and manager in the IT industry. Kaner wrote the book initially as a training and survival guide for his staff, but it soon developed into a bestselling textbook on software testing used by other practitioners throughout the IT industry (Kaner, 1988; Kaner et al., 1999).

Exploration plays an important role in Kaner's approach to software testing. It was soon recognised as an alternative and complementary approach to existing techniques in the literature that emphasise the systematic design and scripting of test cases prior to testing. The notion of ET was further developed by Kaner and other researchers with industry background such as Bach (2002, 2003), Copeland (2004), Whittaker (2009), and Hendrickson (2013). etc. Today, ET is not only widely recognised and practised in the industry, but also has become an active research topic within software testing.

Bach (2003) defines ET as "simultaneous learning, test design, and test execution"; according to Hendrickson (2013) this is widely quoted. Other advocates of ET give similar definitions. Graham et al. (2007, Page 113) defines it as "a test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests". Copeland (2004, Page 202) states that "to the extent that the next test we do is influenced by the result of the last test we did, we are doing exploratory testing. We become more exploratory when we can't tell what tests should be run, in advance of the test cycle". Loveland et al. (2005, Page 339) call ET "artistic testing", defined as "testing that takes early experiences gained with the software and uses them to device new tests not imagined during initial planning. It is often guided by the intuition and investigative instincts of the tester". Whittaker (2009, Page 16) also characterised ET as a process in which "testers may interact with the application in whatever way they want and use the information the application provides to react, change course, and generally explore the application's functionality without restraint". He argued that ET is not ad hoc but a powerful testing technique. The power comes from using the information provided by the software under test to alter the course of testing. This process is what Hendrickson (2013, Page 7) called "steering". Given its importance in ET, Hendrickson (2013) revised Bach's definition by including steering explicitly. She wrote that ET is "simultaneously designing and executing tests to learn about the system, using your insights from the last experiment to inform the next". She further identified four essential elements of ET and explains these distinctive key attributes by regarding ET as experiments as follows.

- *Designing*: identifying interesting things to vary and interesting ways in which to vary them so that the experiment can be better performed.
- *Executing*: all dynamic testing involves executions of the software on test cases, but in ET a test case is executed immediately when it is designed.
- *Learning*: the testers "discover how the software operates".
- *Steering*: using the insights gained from the previous test execution(s) to inform the next.

It is worth noting that "learning", or more precisely, "discovery", is perhaps the most fundamental feature that distinguishes ET from traditional approaches to software testing, which is regarded as a validation and verification technique and/or method; see, for example, Kung and Zhu (2009). Itkonen et al. (2016) regard such traditional approaches to software testing as confirmatory testing. In other words, it aims to confirm existing theories about the software under test, typically to prove (or disprove) the correctness of the software with regards to the expected output and behaviour. They pointed out that ET aims to discover behaviours that are new in contrast to mechanical executions of pre-scripted test cases. Therefore, as Whittaker (2009) pointed out, ET is most suitable for testing software where a precise specification of the system is not available, such as GUI-based systems. Machine learning applications also lack precise specifications so ET is applicable for them as well.

ET is often considered to be a manual testing approach but it need not be. Whittaker (2009) explicitly states that it "doesn't mean we cannot employ automation tools as aids to the process". Itkonen et al. (2016) also point out that the goal of test automation in ET is "to free human resources for other types of testing activities". The goal of this paper is to automate the application of ET in this way when testing machine learning applications.

ET is usually unscripted, whereas traditional testing is scripted as it pre-specifies test cases, mechanically executes them and compares output values to expected values, also pre-specified. However, ET need not be unscripted. Whittaker (2009) pointed out that "It isn't necessary to view exploratory testing as a strict alternative to script-based manual testing. In fact, the two can co-exist quite nicely". He distinguishes four types of ET: freestyle, scenarios-based, strategy-based, and feedback-based (Whittaker, 2009, Page 184). He proposed a set of test strategies as guides to exploratory testers and studied a set of scenarios in exploratory testing. From freestyle to feedback-based ET, the patterns and guides for the testers become more and more specific and prescriptive. However, none of these exploratory strategies have been automated. Our approach to automating ET is to formally define the strategies of exploration as algorithms and then to implement them in the framework of datamorphic testing.

2.3. Datamorphic testing

In the datamorphic software testing method (Zhu et al., 2019a), software artefacts involved in testing are classified into two types: *entities* and *morphisms*.

Test entities are objects and data that are used and/or generated in testing. These include test cases, test suites/sets, the programs under test, and test reports, etc.

Test morphisms are mappings between entities. They generate and transform test entities to achieve testing objectives. They can be implemented as test code and invoked to perform test activities and composed to form test processes. The following are the test morphisms recognised by the datamorphic test tool *Morphy* (Zhu et al., 2020).

- *Test set creators* create sets of test cases. They are called *seed test case makers* in Zhu (2015) and Zhu et al. (2019b). A typical example is random test case generators like fuzzers (Sutton et al., 2007).
- *Datamorphisms* are mappings from existing test cases to new test cases. They are called data mutation operators in the data mutation testing method (Shan and Zhu, 2009).
- *Metamorphisms* are mappings from test cases to Boolean values that assert a program's correctness on test cases. They are test oracles. Formal specifications and metamorphic relations in metamorphic testing (Chen et al., 2018; Segura et al., 2018) can also be used as metamorphisms. *Mutational metamorphic relations* introduced in Zhu (2015) are metamorphisms.
- *Test case metrics* are mappings from test cases to real numbers. They measure test cases giving, for example, the similarity of a test case to the others in the test set.
- *Test case filters* are mappings from test cases to truth values. They can be used, for example, to decide whether a test case should be included in a test set.
- *Test set metrics* are mappings from test sets to real numbers. They measure the test set quality, such as its code coverage (Zhu et al., 1997).
- *Test set filters* are mappings from test sets to test sets. For example, they may remove redundant test cases from a test set for regression testing.
- *Test executers* execute the program under test on test cases and receive the outputs from the program. They are mappings from a piece of program to a mapping from input data to output. That is, they are functors in category theory (Barr and Wells, 1989).
- *Test analysers* analyse test sets and generate test reports. Thus, they are mappings from test sets to test reports.

A *test system* $\mathscr{T} = \langle \mathscr{E}, \mathscr{M} \rangle$ in datamorphic testing consists of a set $\mathscr{E}$ of test entities and a set $\mathscr{M}$ of test morphisms. In Morphy (Zhu et al., 2019a), a test system is specified as a Java class that declares a set of attributes as test entities and a set of methods as test morphisms.

Given a test system, Morphy provides testing facilities to automate testing at three different levels. At the lowest level, various test activities can be performed by invoking test morphisms via a click of buttons on Morphy's GUI. At the medium level, Morphy implements various test strategies to perform complex testing activities through combinations and compositions of test morphisms. At the highest level, test processes are automated by recording, editing and replaying test scripts that consist of a sequence of invocations of test morphisms and strategies.

Test strategies are complex combinations of test morphisms designed to achieve test automation. Three sets of test strategies have been implemented in Morphy:

- *Mutant combination*: combining datamorphisms to generate mutant test cases; see Zhu et al. (2019a).
- *Domain exploration*: searching for the borders between clusters/subdomains of the input space;
- *Test set optimisation*: optimising test sets by employing genetic algorithms.

This paper focuses on domain exploration strategies, which will be defined in Section 4.

2.4. Overview of the proposed approach

The approach of this paper and its previous work (Zhu and Bayley, 2020) is to apply the four ET principles identified previously to test feature-based classifiers built using machine learning and data analytics techniques:

Firstly, on test design, the variations in test cases are formally defined by a set of datamorphisms that can be applied to the features of the classifier under test. These datamorphisms are employed to explore the data space of the ML application. A major contribution of this paper is to formally define the notion of *completeness* for ET test systems, and we prove that complete test systems exist for feature-based classifiers; see Section 3. This enables a complete exploration of the input space.

Secondly, on execution, in our approach, each time a new test case is generated, the ML model is invoked, and the output of the invocation is used to generate the next test case. In fact, the test executor is an important component of our definition of ET test systems; see Section 3.

Thirdly, on learning, our goal in testing is to discover the borders between classes as defined by the ML model under test. Such information is unknown before testing, but the results in the form of Pareto front can improve significantly the tester's knowledge about the behaviour of the model.

Finally, on steering, we study three strategies in which the outputs of previously executed test cases are used in three different ways to decide the next test case. These strategies are defined as algorithms and implemented in the automated datamorphic testing environment Morphy. We will also formally prove that these strategies correctly achieve the goal of exploration, i.e. they detect the borders between classes as defined by the ML model under test; see Section 4.

We will also automate the testing process by implementing the technique in the datamorphic testing framework.

3. Exploratory test systems for feature-based classifiers

Exploratory test systems are test systems for ET. In this section, we will introduce the notion of exploratory test systems and the notion of completeness for such test systems. We will then constructively prove the existence of complete test systems for each type of feature-based classifier.

3.1. Structure of exploratory test system

To apply an exploratory test strategy to a classification program $P : D \to G$ with a distance function $\|\cdot, \cdot\|$, we require that the test system $\mathscr{T} = \langle \mathscr{E}, \mathscr{M} \rangle$ has the following properties.

1. The set $\mathscr{M}$ of morphisms contains a test executer $Exe_P(x)$ that executes the program P under test on a test case x and receives the output of P; that is $Exe_P(x) = P(x)$. In the sequel, we will write $P(x)$ for $Exe_P(x)$ for the sake of simplicity.
2. There is a set $W \subseteq \mathscr{M}$ of unary datamorphisms defined on D. Informally, for each $w \in W$ and $x \in D$, $w(x)$, $w^2(x)$, $\cdots$, $w^n(x)$ can generate a sequence of data points in D, where $w^1(x) = w(x)$, $w^{n+1}(x) = w(w^n(x))$. These datamorphisms are called *traversal methods*.
3. There is also a binary datamorphism $m \in \mathscr{M}$ such that

$$\forall x, y \in D. (\|x, y\| > \delta_m \Rightarrow \|x, m(x, y)\| < \|x, y\| \wedge \|y, m(x, y)\| < \|x, y\|), \quad (1)$$

where $\delta_m = Min_{x \neq y \in D}\{\|x, y\|\}$.
Informally, the datamorphism m calculates a point between x and y, if the distance between them is greater than the minimal distance δ_m among points in the data space. We will call m the *midpoint method*.

Note that for all $x, y \in D$, because the program P under test classifies x and y into different classes, the midpoint $m(x, y)$

between x and y must be either not in the same class as x or not in the same class as y. Formally, we have:

$$(P(x) \neq P(y)) \Rightarrow (P(x) \neq P(m(x,y))) \vee (P(y) \neq P(m(x,y))). \quad (2)$$

Also, note that it is unnecessary to include the distance metric $\|\cdot,\cdot\|$ in the test system as a test morphism. As we will see in Section 4, the algorithms of exploratory test strategies do not need it.

3.2. Completeness of exploratory test systems

For a test system to be able to explore the whole data space of a classifier, we require that the set of datamorphisms is able to reach every data point in the space by applying the datamorphisms on any arbitrary starting point. We say such a set of datamorphisms is *complete*. Completeness may not be possible for a classifier on continuous data space. In such cases, we would like to reach the target point as close as is desired. This property of test system is called *approximate completeness*.

Before we formally define these notions of completeness, we first define the notion of compositions of datamorphisms. Let $\mathscr{M} \neq \emptyset$ be a set of datamorphisms.

Definition 4 (*Composition of Datamorphisms*)**.** Let X be a set of variables ranging over test cases. The set of compositions of datamorphisms in $\mathscr{M}$ is recursively defined as follows.

1. For all $x \in X$, x is a composition of datamorphisms in $\mathscr{M}$ of order 0.
2. $m(e_1, \ldots, e_k)$ is a composition of datamorphisms in $\mathscr{M}$ of order $n+1$, if $m \in \mathscr{M}$ is k-ary, and $e_1, \ldots, e_k$ are compositions of datamorphisms in $\mathscr{M}$, and n is the maximum of the orders of $e_1, \ldots, e_k$. □

Informally, a composition of datamorphisms is an expression with datamorphisms as the operators and variables as the parameters. For example, $m_1(m_2(m_3(x_1, x_2)))$ is a composition of two unary datamorphisms m_1, m_2 and one binary datamorphism m_3, where x_1 and x_2 are variables. Given a composition of datamorphisms, a test case can be obtained by substituting existing test cases for the variables of the composition, and we say that the result is a mutant test case obtained by applying the composition to the existing test cases.

Definition 5 (*Completeness*)**.** An exploratory test system $\mathscr{T} = \langle \mathscr{E}, \mathscr{M} \rangle$ on data space D is *complete*, if for all $a, b \in D$, there is a composition $\phi(x)$ of datamorphisms in $\mathscr{M}$ such that $b = \phi(a)$.

An exploratory test system $\mathscr{T}$ is *approximately complete*, if for all $a, b \in D$ and every $\delta > \delta_m$, there is a composition ϕ of datamorphisms in $\mathscr{M}$ such that $\|b, \phi(a)\| \leq \delta$. □

Note that, in a real-world application, in a multi-dimensional data space some combinations of feature values may be invalid or meaningless. For example, a human who is 2 metres tall but only weights 20 kg is physically impossible. Our completeness requirements on an exploratory test system still require the test system to cover such data. This will enable testing on invalid inputs, which are useful, for example, to understand how the software will react to input errors.

In the remainder of this section, we construct a complete or approximately complete exploratory test system for each type of feature-based classifier.

3.3. Continuous numerical classifiers

Given a continuous numerical classifier, we construct two unary datamorphisms $up_i(x)$ and $down_i(x)$ for each feature f_i as the traversal methods and a binary datamorphism $mid_E(x, y)$ as the midpoint method. The set of datamorphisms will form an approximately complete test system. Let $c_i > 0$ be a given constant real value. We define:

$$up_i(\langle x_1, \ldots, x_K \rangle) = \langle x_1, \ldots, x_i + c_i, \ldots, x_K \rangle \quad (3)$$

$$down_i(\langle x_1, \ldots, x_K \rangle) = \langle x_1, \ldots, x_i - c_i, \ldots, x_K \rangle \quad (4)$$

$$mid_E(\langle x_1, \ldots, x_K \rangle, \langle y_1, \ldots, y_k \rangle) = \left\langle \frac{x_1 + y_1}{2}, \ldots \frac{x_K + y_K}{2} \right\rangle \quad (5)$$

There are many different ways that we can define distance metrics on real numbers. The following is the Euclidean distance on multi-dimensional real numbers.

$$\| \langle x_1, \ldots, x_K \rangle, \langle y_1, \ldots, y_k \rangle \|_E = \sqrt{\sum_{i=1}^{K} (x_i - y_i)^2} \quad (6)$$

The following are a few well-known properties of Euclidean distance, which are useful for proving the approximate completeness of the test system.

Lemma 1. *The distance metrics $\|\cdot,\cdot\|_E$ has the following properties.*

1. *$\forall x \in D.\ \|x, x\|_E = 0$;*
2. *$\forall x, y \in D.\ \|x, y\|_E \geq 0$;*
3. *$\forall x, y \in D.\ \|x, z\|_E < \|x, y\|_E \wedge \|z, y\|_E < \|x, y\|_E$, where $z = mid_E(x, y)$.*
4. *$\forall x, y \in D.\ \|x, z\|_E = \frac{\|x,y\|_E}{2}$, where $z = mid_E(x, y)$.* □

Let $W_E = \{up_i(x) \mid i = 1, \ldots, K\} \cup \{down_i(x) \mid i = 1, \ldots, K\} \cup \{mid(x, y)\}$. Applying these properties of the midpoint datamorphism $mid_E(x, y)$ and Euclidean distance metrics $\|x, y\|_E$, we can prove that the set of datamorphisms W_E defined above satisfies the requirements of exploratory test systems on datamorphisms.

Theorem 1. *The set W_E of datamorphisms together with the distance metrics $\|x, y\|_E$ satisfy the conditions of exploratory test systems on datamorphisms.*

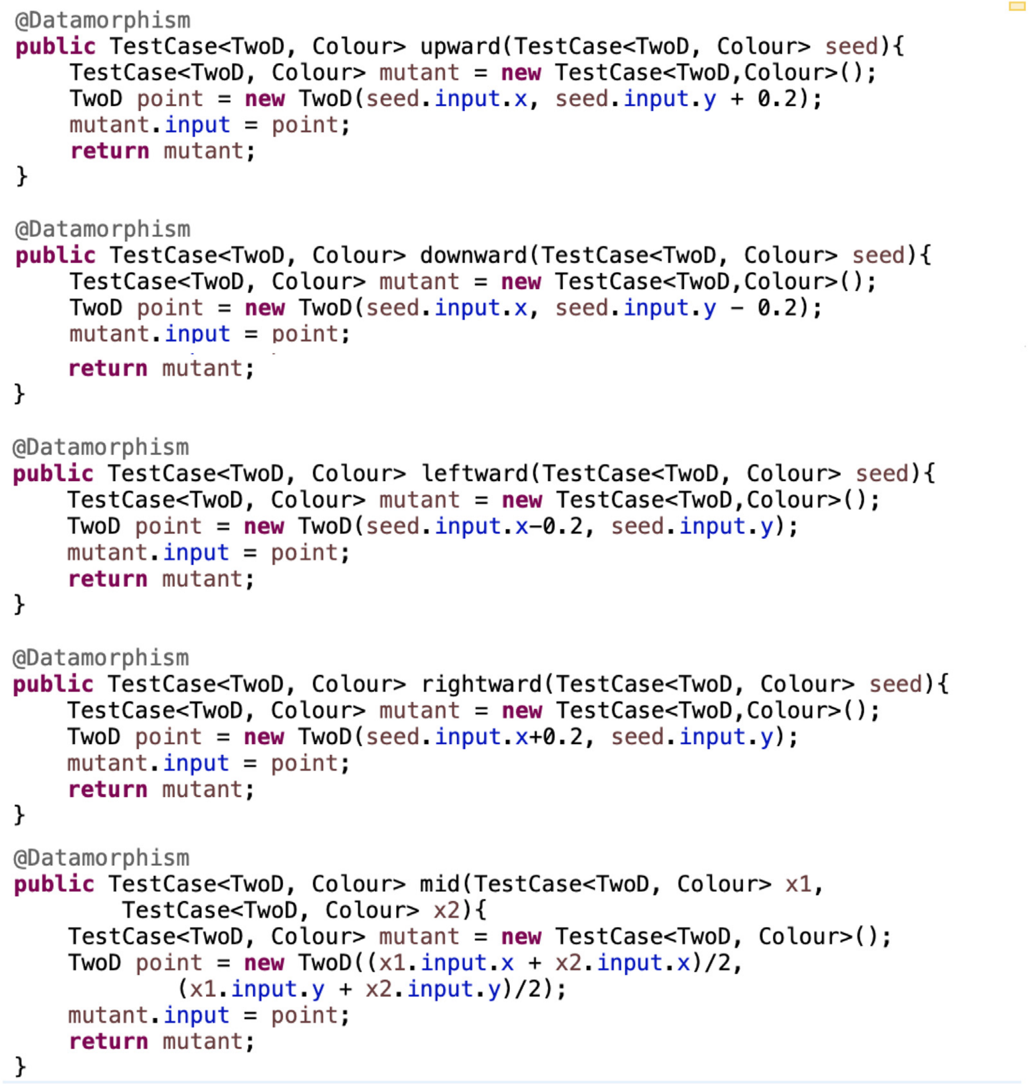
Proof. By (6), $\delta_m = Min_{x \neq y \in D}(\|x, y\|_E) = 0$. Therefore, by Lemma 1(4), the condition given in Eq. (1) is true. The theorem is true. □

Example 2. Fig. 2 gives the traversal and midpoint methods in the Morphy test specification for the classifier of the running example. The *leftward* and *rightward* methods implement the traversal methods $down_x$ and up_x, respectively. The *upwards* and *downward* methods implement the traversal methods up_y and $down_y$, respectively, where $c_x = c_y = 0.2$. The method *mid* implements the mid_E datamorphism, which calculates the geometric midpoint between x and y as defined in Eq. (5). Therefore, by Theorem 1, they form an exploratory test system with the following distance function.

$$\| \langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle \| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \quad □$$

The following theorem states that W_E is approximately complete.

Theorem 2. *The set W_E of datamorphisms is approximately complete for a continuous numerical feature-based classifier P defined on the data space $D = D_1 \times \cdots \times D_K$, $K > 0$.*



```
@Datamorphism
public TestCase<TwoD, Colour> upward(TestCase<TwoD, Colour> seed){
    TestCase<TwoD, Colour> mutant = new TestCase<TwoD,Colour>();
    TwoD point = new TwoD(seed.input.x, seed.input.y + 0.2);
    mutant.input = point;
    return mutant;
}

@Datamorphism
public TestCase<TwoD, Colour> downward(TestCase<TwoD, Colour> seed){
    TestCase<TwoD, Colour> mutant = new TestCase<TwoD,Colour>();
    TwoD point = new TwoD(seed.input.x, seed.input.y - 0.2);
    mutant.input = point;
    return mutant;
}

@Datamorphism
public TestCase<TwoD, Colour> leftward(TestCase<TwoD, Colour> seed){
    TestCase<TwoD, Colour> mutant = new TestCase<TwoD,Colour>();
    TwoD point = new TwoD(seed.input.x-0.2, seed.input.y);
    mutant.input = point;
    return mutant;
}

@Datamorphism
public TestCase<TwoD, Colour> rightward(TestCase<TwoD, Colour> seed){
    TestCase<TwoD, Colour> mutant = new TestCase<TwoD,Colour>();
    TwoD point = new TwoD(seed.input.x+0.2, seed.input.y);
    mutant.input = point;
    return mutant;
}

@Datamorphism
public TestCase<TwoD, Colour> mid(TestCase<TwoD, Colour> x1,
        TestCase<TwoD, Colour> x2){
    TestCase<TwoD, Colour> mutant = new TestCase<TwoD, Colour>();
    TwoD point = new TwoD((x1.input.x + x2.input.x)/2,
            (x1.input.y + x2.input.y)/2);
    mutant.input = point;
    return mutant;
}
```

Fig. 2. Datamorphisms of the running example.

Proof. We prove that for any given points $a = \langle a_1, \dots, a_K \rangle$, $b = \langle b_1, \dots, b_K \rangle \in D$ and $\delta > 0$, we can construct a composition ϕ of datamorphisms such that $\|b, \phi(a)\|_E < \delta$. The composition $\phi(x)$ is defined as follows.

$$\phi(x) = m^{n_\delta} \circ ud_1^{n_1} \circ \cdots \circ ud_K^{n_K}(x), \tag{7}$$

where

$$m(x) = mid_E(b, x), \qquad ud_i^{n_i}(x) = \begin{cases} up_i^{n_i}(x) & \text{if } a_i \geq b_i \\ down_i^{n_i}(x) & \text{if } a_i < b_i \end{cases},$$

$$n_i = \lfloor \frac{|a_i - b_i|}{c_i} \rfloor, \qquad n_\delta = \lceil ln(\frac{c}{\delta}) \rceil, \qquad c = \sqrt{\sum_{n=1}^{K} c_i^2}.$$

Note that $ud_i(x)$ is either $up_i(x)$ or $down_i(x)$ depending on whether the i'th element of a is greater than the i'th element of b.

Let $a' = ud_1^{n_1} \circ \cdots \circ ud_K^{n_K}(x) = \langle a'_1, \dots, a'_K \rangle$. We have that a' is obtained by applying $ud_i(x)$ for n_i times on a, for $i = 1, \dots, K$. The i'th element of a' will be $a'_i = a_i \pm n_i \cdot c_i$ by the definition of datamorphisms $up_i(x)$ and $down_i(x)$. By the definition of n_i, we have that $|b_i - a'_i| < c_i$, for all $i = 1, \dots, K$. Therefore,

$$\|b, a'\| = \sqrt{\sum_{n=1}^{K} (b_i - a'_i)^2} \leq \sqrt{\sum_{n=1}^{K} c_i^2} = c.$$

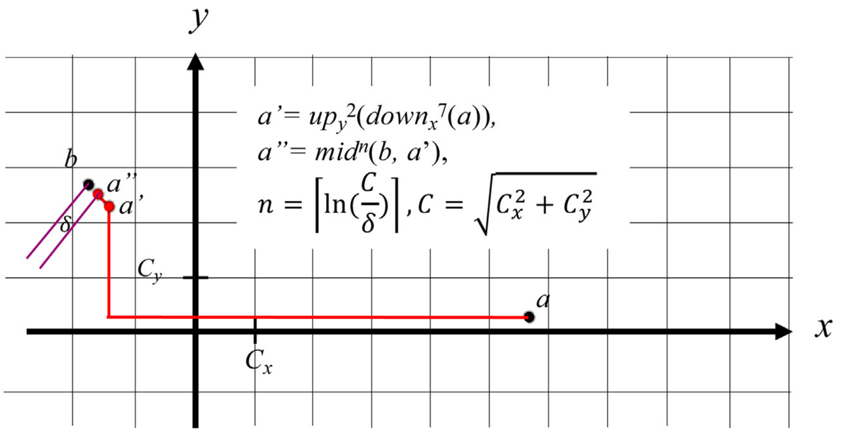
Applying $m(x)$ on a' for n times, we get $a'' = m^n(a')$. By Lemma 1(4), we have that $\|b, a''\| = \|b, a'\|/2^n \leq c/2^n$. Therefore, when $n \geq ln(\frac{c}{\delta})$, we have that $\|b, a''\| \leq \delta$. The theorem follows immediately that $n_\delta = \lceil ln(\frac{c}{\delta}) \rceil \geq ln(\frac{c}{\delta})$. □

Example 3. The exploratory test system given in Example 2 is approximately complete, because for all points a, b in the data space and $\delta > 0$, we have a composition $\phi(x)$ of datamorphisms such that $\|b, \phi(a)\| \leq \delta$; see Fig. 3 for an illustration of how to construct the composition of datamorphisms. □

3.4. Discrete non-numerical classifiers

If the classifier P is a discrete non-numerical feature-based classifier then for each $i = 1, \dots, K$, D_i is a non-empty finite set. Let $D_i = \{v_{i,1}, v_{i,2}, \dots, v_{i,n_i}\}$, where $n_i > 0$. We define two unary datamorphisms $up_i(x)$ and $down_i(x)$ as the traversal methods as follows.

$$up_i(\langle x_1, \dots, x_K \rangle) = \langle x_1, \dots, x'_i, \dots, x_K \rangle,$$



y
x
$a' = up_y^2(down_x^7(a))$,
$a'' = mid^n(b, a')$,
$n = \left\lceil \ln(\frac{C}{\delta}) \right\rceil, C = \sqrt{C_x^2 + C_y^2}$
b
a''
a'
δ
C_y
C_x
a

Fig. 3. Construction of the walk path in the running example.

$$\text{where } x_i' = \begin{cases} v_{i,j+1} & \text{if } x_i = v_{i,j} \text{ and } j < n_i \\ v_{i,n_i} & \text{if } x_i = v_{i,n_i} \end{cases} \tag{8}$$

$$down_i(\langle x_i, \ldots, x_K \rangle) = \left\langle x_i, \ldots, x_i', \ldots, x_K \right\rangle,$$

$$\text{where } x_i' = \begin{cases} v_{i,j-1} & \text{if } x_i' = v_{i,j} \text{ and } j > 1 \\ v_{i,1} & \text{if } x_i' = v_{i,1} \end{cases} \tag{9}$$

Let $x, y \in D$, $x = \langle x_1, \ldots, x_K \rangle$ and $y = \langle y_1, \ldots, y_K \rangle$. The distance between x and y, written $\|x, y\|_D$, is defined as the number of elements in x and y that are different. Let $\Delta(x, y) = \langle d_1, \ldots, d_k \rangle$, $0 \leq k \leq K$, be the sequence of elements in x that are different from the corresponding elements in y. Therefore, we have that $\|x, y\|_D = k$.

The following Lemma states that the function $\|\cdot,\cdot\|_D : D \times D \to N$ satisfies the conditions of distance metrics. The proof is straightforward, and thus is omitted for the sake of space.

Lemma 2. *The function $\|\cdot,\cdot\|_D : D \times D \to N$ defined above satisfies the conditions of distance metrics. That is, for all $x, y, z \in D$, we have that $\|x, x\|_D = 0$, $\|x, y\|_D \geq 0$, $\|x, y\|_D = \|y, x\|_D$, and $\|x, y\|_D + \|y, z\|_D \geq \|x, z\|_D$.* □

We now define a binary datamorphism $mid_D(x, y)$ as the midpoint method as follows.

$$mid_D(x, y) = \langle z_1, \ldots, z_K \rangle, \tag{10}$$

where

$$z_i = \begin{cases} x_i & \text{if } x_i = y_i \\ x_i & \text{if } x_i \neq y_i \text{ and } x_i \text{ is an odd-indexed element in } \Delta(x, y) \\ y_i & \text{if } x_i \neq y_i \text{ and } x_i \text{ is an even-indexed element in } \Delta(x, y) \end{cases} \tag{11}$$

The following theorem gives some useful special properties of the distance metrics $\| \ \|_D$ and midpoint datamorphism mid_D on discrete data space. These properties are easy to prove by using the definitions of the distance function and discrete non-numerical data space. Details are omitted for the sake of space.

Lemma 3. *For all $x, y \in D$, we have that*

1. $x \neq y \Rightarrow \|x, y\|_D \geq 1$;
2. $\|x, y\|_D \leq K$;
3. $mid_D(x, x) = x$;
4. $\|x, y\|_D = 1 \Rightarrow (mid_D(x, y) = x) \vee (mid_D(x, y) = y)$;
5. $\|x, y\|_D > 1 \Rightarrow \|x, z\|_D < \|x, y\|_D \wedge \|z, y\|_D < \|x, y\|_D$, *where* $z = mid_D(x, y)$. □

Let $W_D = \{up_i(x) \mid i = 1, \ldots, K\} \cup \{down_i(x) \mid i = 1, \ldots, K\} \cup \{mid_D(x, y)\}$.

Theorem 3. *W_D and the distance metrics $\|\cdot,\cdot\|_D$ together satisfy the requirements of exploratory test systems on datamorphisms.*

Proof. By Lemma 3(1), $\delta_m = Min_{x \neq y \in D}\{\|x, y\|_D\} = 1$. By Lemma 3(5), $mid_D(x, y)$ and $\|x, y\|_D$ meet the condition on the midpoint method given in Eq. (1). Thus, the theorem is true. □

The following theorem states that the set of datamorphisms constructed above is complete.

Theorem 4. *The set W_D of datamorphisms is complete for a discrete non-numerical feature-based classifier P defined on the data space $D = D_1 \times \cdots \times D_K$, $K > 0$.*

Proof. For any given points $a, b \in D$, we construct a composition of datamorphisms $\phi(x)$ such that $\phi(a) = b$. We define

$$\phi(x) = ud_1^{n_1} \circ \cdots \circ ud_K^{n_K}(x), \tag{12}$$

where

$$ud_i^{n-i}(x) = \begin{cases} up_i^{n_i}(x) & \text{if } a_i = v_{i,c_a} \in D_i, \\ & \quad b_i = v_{i,c_b} \in D_i, \text{and } c_b \geq c_a \\ down_i^{n_i}(x) & \text{if } a_i = v_{i,c_a} \in D_i, \\ & \quad b_i = v_{i,c_b} \in D_i \text{ and } c_b < c_a \end{cases},$$

$$n_i = |c_x - c_y|. \tag{13}$$

By (13), $ud_i(x)$ is either $up_i(x)$ or $down_i(x)$ depending on the difference between a and b on the i'th element, and n_i is the distance between the i'th elements of a and b. By the definitions of $up_i(x)$ and $down_i(x)$, we have that $\phi(a) = b$. Therefore, by Definition 5 of completeness, the theorem is true. □

3.5. Discrete numerical classifiers

For a discrete numerical classifier, we also define two unary datamorphisms $up_i(x)$ and $down_i(x)$ for each feature f_i as the traversal methods. The $up_i(x)$ datamorphism on feature f_i is defined as follows.

$$up_i(\langle x_1, \ldots, x_K \rangle) = \left\langle x_1, \ldots, x_i', \ldots, x_K \right\rangle, \text{ where } x_i' = x_i + 1. \tag{14}$$

The datamorphism $down_i(x)$ is defined as follows.

$$down_i(\langle x_1, \ldots, x_K \rangle) = \left\langle x_1, \ldots, x_i', \ldots, x_K \right\rangle \tag{15}$$

where $x_i' = x_i - 1$, if the dataset D_i is the set of integer values; and $x_i' = \begin{cases} x_i - 1 & \text{if } x_i > 0 \\ 0 & \text{if } x_i = 0 \end{cases}$, if the dataset D_i is the set of natural numbers.

The midpoint datamorphism $mid_N(x, y)$ is defined as follows.

$$mid_N(\langle x_1, \dots, x_K\rangle, \langle y_1, \dots, y_K\rangle) = \left\langle \lfloor \frac{|x_1 - y_1|}{2} \rfloor, \dots, \lfloor \frac{|x_K - y_K|}{2} \rfloor \right\rangle \quad (16)$$

Now, we define the distance metric $\|\cdot, \cdot\|_N$ on the data space, as follows.

$$\| \langle x_1, \dots, x_K\rangle, \langle y_1, \dots, y_K\rangle \|_N = \sum_{i=1}^{K} |y_i - x_i| \quad (17)$$

Similar to Lemma 2, we can prove that the function $\|\cdot, \cdot\|_N : D \times D \to N$ satisfies the conditions of distance metrics. The proof is straightforward, and thus is omitted for the sake of space.

Lemma 4. *The function $\|\cdot, \cdot\|_N : D \times D \to N$ defined above satisfies the condition of distance metrics. That is, for all $x, y, z \in D$, we have that $\|x, x\|_N = 0$, $\|x, y\|_N \geq 0$, $\|x, y\|_N = \|y, x\|_N$, and $\|x, y\|_N + \|y, z\|_N \geq \|x, z\|_N$.* □

The midpoint datamorphism $mid_N(x, y)$ and the distance metrics $\|x, y\|_N$ have the following properties. Again, they are easy to prove by the definitions of the distance function and discrete numerical data space. Details are omitted for the sake of space.

Lemma 5. *For all $x, y \in D$, we have that*

1. $x \neq y \Rightarrow \|x, y\|_N \geq 1$;
2. $mid_N(x, x) = x$;
3. $\|x, y\|_N = 1 \Rightarrow (mid_N(x, y) = x) \vee (mid_N(x, y) = y)$;
4. $\|x, y\|_N > 1 \Rightarrow \|x, z\|_N < \|x, y\|_N \wedge \|z, y\|_N < \|x, y\|_N$, *where $z = mid(x, y)$.* □

Let $W_N = \{up_i(x) \mid i = 1, \dots, K\} \cup \{down_i(x) \mid i = 1, \dots, K\} \cup \{mid(x, y)\}$. The following theorem states that the set W_N of datamorphisms constructed above satisfies the conditions of exploratory test systems. The proof is very similar to that of Theorem 3 so the details are omitted for the sake of space.

Theorem 5. *W_N and the distance metrics $\|\cdot, \cdot\|_N$ together satisfy the requirements of exploratory test systems on datamorphisms.* □

The following theorem states that the set W_N of datamorphisms constructed above is complete.

Theorem 6. *The set W_N of datamorphisms is complete for a discrete numerical feature-based classifier P defined on the data space $D = D_1 \times \cdots \times D_K$, $K > 0$,*

Proof. For any given points $a, b \in D$, we construct a composition $\phi(x)$ of datamorphisms such that $\phi(a) = b$. We define

$$\phi(x) = ud_1^{n_1} \circ \cdots \circ ud_K^{n_K}(x), \quad (18)$$

where

$$ud_i^{n_i}(x) = \begin{cases} up_i^{n_i}(x) & \text{if } a_i \geq b_i \\ down_i^{n_i}(x) & \text{if } a_i < b_i \end{cases}, \quad n_i = |a_i - b_i|. \quad (19)$$

By (19), $ud_i(x)$ is either $up_i(x)$ or $down_i(x)$ depending on the difference between a and b on the i'th element, and n_i is the absolute value of the distance between the i'th elements of a and b. By the definitions of $up_i(x)$ and $down_i(x)$, we have that $\phi(x) = b$. Therefore, by Definition 5 of completeness, the theorem is true. □

3.6. Hybrid feature-based classifiers

Let $P : D \to C$ be a hybrid feature-based classifier. Without lost of generality, we assume that $D = D_1 \times \cdots \times D_u \times N_1 \times \cdots \times N_v \times R_1 \times \cdots \times R_w$, where $D_1, \dots, D_u$ are discrete non-numerical features, $N_1, \dots, N_v$ are discrete numerical features, and $R_1, \dots, R_w$ are continuous numerical features, and at least two of u, v and w are greater than zero.

We now define unary datamorphisms $up_i(x)$ and $down_i(x)$ as the traversal methods as follows.

1. If feature f_i is discrete non-numerical, we use Eq. (8) to define $up_i(x)$.
2. If feature f_i is discrete numerical, we use Eq. (14) to define $up_i(x)$.
3. If feature f_i is continuous numerical, we use Eq. (3) to define $up_i(x)$.

Similarly, we define $down_i(x)$ depending on the type of features and using the Eqs. (9), (15) and (4), accordingly.

Before we formally define a binary datamorphism as the midpoint method and a distance metric, let us first introduce some notation.

Let $x = \langle d_1, \dots, d_u, n_1, \dots, n_v, r_1, \dots, r_w\rangle \in D$. We write $x_D = \langle d_1, \dots, d_u\rangle$, $x_N = \langle n_1, \dots, n_v\rangle$, and $x_E = \langle r_1, \dots, r_w\rangle$. We also write $x = x_D \oplus x_N \oplus x_E$. In general, $\oplus$ is an operator on vectors defined as follows.

$$\langle x_1, \dots, x_n\rangle \oplus \langle y_1, \dots, y_m\rangle = \langle x_1, \dots, x_n, y_1, \dots, y_m\rangle$$

Now, we define a binary datamorphism $mid_H(x, x')$ as follows.

$$mid_H(x, x') = mid_D(x_D, x'_D) \oplus mid_N(x_N, x'_N) \oplus mid_E(x_E, x'_E) \quad (20)$$

We now define $\|\cdot, \cdot\|_H : D \times D \to \mathbb{R}^+$ as follows.

$$\|x, x'\|_H = \|d, d'\|_D + \|n, n'\|_N + \|r, r'\|_E. \quad (21)$$

The following lemma states that the above equation defines a distance metric. It follows immediately the properties of $\|\cdot, \cdot\|_D$, $\|\cdot, \cdot\|_N$ and $\|\cdot, \cdot\|_E$. Details are omitted.

Lemma 6. *Function $\|\cdot, \cdot\|_H$ satisfies the conditions of distance metrics.* □

Let $W_H = \{up_i(x)\}_i \cup \{down_i(x)\}_i \cup \{mid_H(x)\}$, where $up_i(x)$, $down_i(x)$ and $mid_H(x, y)$ are defined as above.

Theorem 7. *The set of datamorphisms W_H and the distance metrics $\|x, y\|_H$ together satisfy the conditions of exploratory test systems.*

Proof. First, from the definition of $\|x, y\|_H$, we have that $\delta_m = Min_{x \neq y \in D}\{\|x, y\|_H\}$. If there is at least one feature in the data space D that is a continuous numerical feature, then it is easy to see that $\delta_m = 0$. Otherwise, all features are either discrete non-numerical or discrete numerical so we have $\delta_m = 1$.

Second, let $x, y \in D$ and $\|x, y\|_H > \delta_m$, and $z = mid_H(x, y)$. By the definitions of $mid_H(x, y)$ and $\|x, y\|_H$, we have that

$$\begin{aligned} \|x, z\|_H &= \|x_D, z_D\|_D + \|x_N, z_N\|_N + \|x_E, z_E\|_E \\ &= \|x_D, mid_D(x_D, y_D)\|_D + \|x_N, mid_N(x_N, y_N)\|_N + \|x_E, mid_E(x_E, y_E)\|_E \\ &< \|x_D, y_D\|_D + \|x_N, y_N\|_N + \|x_E, y_E\|_E \\ &= \|x, y\|_H \end{aligned}$$

Similarly, we have $\|y, z\|_H < \|x, y\|_H$. Therefore, the theorem is true. □

Theorem 8. *Let P be a hybrid feature-based classifier, and W_H be the set of datamorphisms defined above.*

1. *If there is a continuous numerical feature in the data space of P, W_H is approximately complete.*
2. *If there is no continuous numerical feature in the data space of P, the set W_H of datamorphisms is complete.*

Proof. Similar to the proofs of Theorems 4, 6 and 2, for any given points a and b in the data space, and any given real number $\delta > 0$, we construct a composition $\phi(x)$ of datamorphisms such that $\|b, \phi(a)\|_H \leq \delta$.

Let $a = a_D \oplus a_N \oplus a_E$ and $b = b_D \oplus b_N \oplus b_E$.

By the proof of Theorem 4, there is a composition of datamorphisms $\phi_D(x)$ such that $b_D = \phi_D(a_D)$.

By Theorem 6, there is a composition $\phi_N(x)$ of datamorphisms such that $b_n = \phi_N(a_N)$.

By Theorem 2, there is a composition $\phi_E(x)$ of datamorphisms such that $\|b_E, \phi_E(a_E)\| \leq \delta$.

By the definition of the datamorphisms for hybrid feature-based classifier, $\phi_D(x)$, $\phi_N(x)$ and $\phi_E(x)$ are also compositions of the datamorphisms in W_H. Therefore, $\phi(x) = \phi_E \circ \phi_N \circ \phi_D(x)$ is a composition of datamorphisms in W_H.

Let $a' = \phi(a)$. It is easy to see that $\phi(a) = \phi_D(a_D) \oplus \phi_N(a_N) \oplus \phi_E(a_E)$. Therefore,

$$\begin{aligned}
&\|b, a'\|_H = \|b, \phi(a)\|_H \\
&= \|b_D \oplus b_N \oplus b_E, \phi_D(a_D) \oplus \phi_N(a_N) \oplus \phi_E(a_E)\|_H \\
&= \|b_D, \phi_D(a_D)\|_D + \|b_N, \phi_N(a_N)\|_N + \|b_E, \phi_E(a_E)\|_E \\
&= 0 + 0 + \|b_E, \phi_E(a_E)\|_E \leq \delta
\end{aligned}$$

Therefore, statement (2) of the theorem is true.

If there is no continuous numerical feature in the data space, i.e. b_E and a_E are empty, then $\|b_E, \phi_E(a_E)\|_E = 0$. Therefore, in such a case, statement (1) is true. □

4. Exploration strategies

This section presents the algorithms of three different exploratory strategies for testing clustering and classification applications. We also prove their correctness and illustrate their behaviour by using the running example given in the previous section.

4.1. Random target strategy

Let us start with a simple exploration strategy based on random selection of two test cases in order to find the Pareto front of the classification groups between these two test cases. We call this strategy *random target strategy*.

The strategy starts by selecting a pair of two test cases x and y at random. If the outputs of the program P under test on these test cases are different, i.e. $P(x) \neq P(y)$, then a point z_1 between x and y is generated by using the binary datamorphism of the midpoint method $mid(x, y)$, i.e. $z_1 = mid(x, y)$. The program P is executed on this mutant test case z_1 to classify it. The classification of z_1 must be different from one of the original pair of test cases; say $P(z_1) \neq P(x)$. Thus, we can repeat the above steps with x and z_1 as the pair of test cases, and a further mutant z_2 can be generated. This process is repeated a number of times to ensure the distance between the final pair of points is small enough. See Algorithm 1.

Let $n > 0$ be any given natural number. We write $RT(n) = \langle a, b\rangle$ to denote the results of executing Algorithm 1 with n as the parameter *steps* and $\langle a, b\rangle$ as the output.

Assume that the exploratory test system has the following properties.

1. There is a constant $c > 1$ such that
$$\forall x, y \in D. \left(\frac{Max\{\|x, z\|, \|z, y\|\}}{\|x, y\|}\right) \leq 1/c, \quad (22)$$
where $z = mid(x, y)$.
2. There is a constant $d_m > 0$ such that
$$\forall x, y \in D.(\|x, y\| \leq d_m). \quad (23)$$

Algorithm 1 (Random Target Strategy)

```
Input: testSet: Test Pool; steps: Integer; mid(x, y): Binary datamorphism;
Output: a, b: Test Case;
Begin
  1: Select two different test cases x and y in testSet at random;
  2: Execute program P on test cases x and y;
  3: Check if a pair of the Pareto front exists between x to y:
  if (x.output = y.output) then return ⟨null, null⟩
  end if
  4: Refinement:
  for i ← 1 to steps do
     z = mid(x, y);
     if (x.output ≠ z.output) then  y = z
     else x = z;
     end if
  end for;
  a = x; b = y;
  return ⟨a, b⟩;
End
```

Then, we have the following theorem about the correctness of the random target strategy algorithm.

Theorem 9. *If $RT(n) = \langle a, b\rangle \neq \langle null, null\rangle$, then $\langle a, b\rangle$ is a pair in the Pareto front according to P with respect to $\|\cdot, \cdot\|$ and δ, if $d_m/c^n < \delta$.*

Proof. If $RT(n) = \langle a, b\rangle \neq \langle null, null\rangle$ then the condition of the If-statement in step (3) is **false**. Thus, the loop is executed. It is easy to see that the For-loop in *Step 4* in the algorithm terminates.

We now prove that the following is a loop invariant by induction on the number i of iterations of the loop body.

$$\|x, y\| \leq \frac{d_m}{c^i} \wedge P(x) \neq P(y).$$

When entering the loop, by assumption (23), the distance between the data points stored in variable x and y satisfies the following inequality.

$$\|x, y\| \leq d_m$$

Since the condition of the If-statement is false, we have that

$$P(x) = x.output \neq y.output = P(y).$$

Therefore, the loop invariant is true for $i = 0$.

Assume that the loop invariant is true for $i = n \geq 0$.

After the execution of the loop body one more time (i.e. $i = n+1$), by applying the Hoare logic of the If-statements in the loop body, the distance d'_x between the data points stored in variables x and y will become either $\|x, z\|$ or $\|z, y\|$, where $z = mid(x, y)$. By assumption (22), in both cases we have that

$$d'_x \leq Max\{\|x, z\|, \|z, y\|\} \leq \|x, y\|/c \leq d_m/c^{n+1}.$$

By the condition of the If-statement in the loop body and the property (2), applying Hoare logic we have that, after the execution of the loop body, the data points stored in variables x and y have the property that $P(x) \neq P(y)$. Therefore, the condition is a loop invariant according to Hoare logic.

When the loop exits, $i = steps = n$. By Hoare logic, after executing the assignment statements $a = x$ and $b = y$, we have that

$$\|a, b\| \leq d_m/c^n \wedge P(a) \neq P(b).$$

Therefore, the theorem is true by Definition 1. □

The algorithm of random target strategy can be run multiple times to generate a number of pairs for the Pareto front.

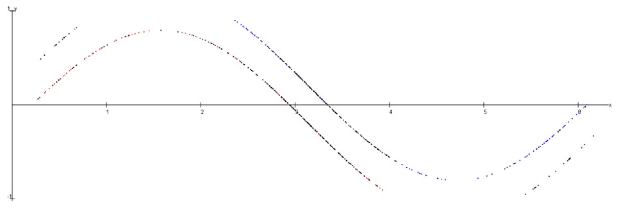


Fig. 4. Pareto front generated by random target.

Example 4. For example, applying the random target strategy to the running example, we can obtain a test set shown in Fig. 4 when 1000 pairs of test cases are selected at random from a test set of 300 random test cases. A total of 641 pairs of Pareto front test cases were generated. The success rate in generating a pair for the Pareto front is 64.1%. The set of Pareto front pairs shows clearly the boundary between the subdomains classified by the software.

In this example, the number of steps n is 20. Since the data space $D = [0, 2\pi] \times [-1, 1]$, if the distance function $\|x, y\|$ is $Eucl(x, y)$, we have that $d_m = 2\sqrt{\pi^2 + 1}$. By the definition of $mid(x, y)$, we have that

$$\frac{Max(\{\|x, z\|, \|y, z\|\})}{\|x, y\|} = 1/2.$$

So, $c = 2$. By Theorem 9, for the distance δ between each pair in the Pareto front, we have that

$$\delta \leq \frac{d_m}{c^{20}} = \frac{\sqrt{\pi^2 + 1}}{2^{19}}.$$

Note that the pairs of test cases in the Pareto front are so close together that they are visually indistinguishable. □

4.2. Directed walk strategy

A variation of the random target strategy is to start with one test case (rather than a pair) and apply a unary datamorphism repeatedly until a test case of different classification is found. Then, the Pareto front between these two test cases is searched for in the same way as for the random target strategy. In this strategy, the unary datamorphism (i.e. a mutation operator) is the traversal method. The repeated application of the mutation operator makes a 'walk' in one direction until a test case in a different class is found or too many iterations have been carried out and the exploration has gone too far.

Note that, a walk in one direction may not be able to find a data point in a different class. In that case, the algorithm returns $\langle null, null \rangle$. Let $m, n > 0$ be any given natural numbers. We write $DW(m, n) = \langle a, b \rangle$ to denote the results of executing Algorithm 2 with m as the walking distance and n as the number of *steps* and $\langle a, b \rangle$ as the output. Assume that the exploratory test system satisfies assumption (22) and has the following property.

There is a constant $d_s > 0$ such that

$$\forall x \in D. (\|x, d(x)\| \leq d_s) . \tag{24}$$

where d_s is called the step size of the traversal method $d(x)$. Then, we have the following correctness theorem for the directed walk algorithm.

Theorem 10. *If $DW(m, n) = \langle a, b \rangle \neq \langle null, null \rangle$ then $\langle a, b \rangle$ is a pair in the Pareto front according to P with respect to $\|\cdot, \cdot\|$ and δ, if $d_s/c^n < \delta$, where n is the number of steps.*

Proof. If $DW(m, n) = \langle a, b \rangle \neq \langle null, null \rangle$, then the condition of the If-statement in step (4) is **false**. Thus, the For-loop of Step (5)

Algorithm 2 (Directed Walk)

```
Input: TestSet: test set; walkDistance: integer; steps: Integer;
  d(x): Unary datamorphism; mid(x, y): Binary datamorphism;
Output: a, b: Test Case;
Begin
  1: Select a test cases x in testSet at random;
  2: Execute program P on test case x;
  3: Walk in one direction as follows:
  Bool found = false;
  for $i \leftarrow 1$ to walkingDistance do
     $y = d(x)$;
     Execute software on test case y;
     if ($x.output \neq y.output$) then found = true; break;
     else $x = y$;
     end if
  end for
  4: Check if a Pareto front can be found:
  if ($\neg found$) then return $\langle null, null \rangle$;
  end if
  5: Refinement:
  for $i \leftarrow 1$ to steps do
     $z = mid(x, y)$;
     if ($x.output \neq z.ouptut$) then  y = z;
     else x = z;
     end if;
  end for
  $a = x$; $b = y$;
  return $\langle a, b \rangle$;
End
```

is executed. It is easy to see that the For-loop in *Step 5 Refinement* in the algorithm terminates.

Similar to the proof of Theorem 9, by the definition of d_s and assumption (24), the following is a loop invariant of the loop by induction on the number i of iterations of the loop body.

$$\|x, y\| \leq \frac{d_s}{c^i} \wedge P(x) \neq P(y).$$

When the loop exits, $i = steps = n$. By Hoare logic, after executing the assignment statements $a = x$ and $b = y$, we have that

$$\|a, b\| \leq d_s/c^n \wedge P(a) \neq P(b).$$

Therefore, the theorem is true by Definition 1. □

Example 5. For example, starting from 1000 random test cases using the directed walk strategy with the $upward(x)$ datamorphism as the unary traversal method, a set of 161 Pareto front pairs were generated; shown in Fig. 5. The set of Pareto front pairs also shows clearly parts of the boundaries between classes. The success rate of finding a pair of Pareto front on one test case is 16.1%.

In this example, the number n of steps is also 20. By the definition of $upward(x)$ traversal method, we have that $d_s = 0.2$, if the distance function $\|x, y\|$ is $Eucl(x, y)$. As in Example 4, by the definition of $mid(x, y)$, we have that $c = 2$. By Theorem 10, for the distance δ between each Pareto front pair, we have that

$$\delta \leq \frac{d_s}{c^{20}} = 0.2 \times \frac{1}{2^{20}}.$$

Again, the distance between the test cases in each Pareto front pair is so small that they are not visually distinguishable, so they appear as one dot in Fig. 5. □

4.3. Random walk strategy

If multiple traversal methods are available, a random walk can be performed by selecting the direction of the next step at

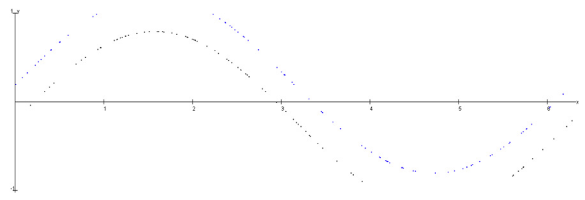


Fig. 5. Pareto fronts generated by directed walk.

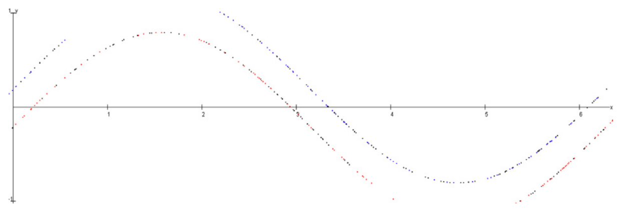


Fig. 6. The Pareto fronts generated by random walk.

random. This is similar to the random walk testing in a web GUI hyperlink test. The algorithm is given below.

Algorithm 3 (Random Walk Strategy)

```
Input: testSet: Test Set; walkingDistance: Integer; steps: Integer;
  $d_1(x), \cdots, d_k(x)$: Unary datamorphism ($k > 1$); $mid(x, y)$: Binary
  datamorphism;
Output: a, b: Test Case;
Begin
  1: Select a test case x in testSet at random;
  2: Execute program P on test case x;
  3: Walking at random to search for test case in a different class:
  Bool found = false;
  for $i \leftarrow 1$ to walkingDistance do
      Get a random integer r in the range $[1, k]$
      $y = d_r(x)$;
      Execute program P on test case y;
      if ($x.output \neq y.output$) then found = true; break;
      else x=y;
      end if
  end for
  4: Check if a Pareto front can be found:
  if ($\neg found$) then  return $\langle null, null \rangle$;
  end if
  5: Refinement:
  for $i \leftarrow 1$ to steps do
      $z = mid(x, y)$;
      if ($x.output \neq z.ouptut$) then  $y = z$;
      else x = z;
      end if
  end for
  $a = x$; $b = y$;
  return $\langle a, b \rangle$;
End
```

We write $RW(m, n) = \langle a, b \rangle$ to denote the results of executing Algorithm 3 with m as the walking distance and n as the *steps* and $\langle a, b \rangle$ as the output. Assume that the exploratory test system satisfies assumption (22) and has the following property. There is a constant $d_{sm} > 0$ such that

$$\forall x \in D. \forall d_i \in W. (\|x, d_i(x)\| \leq d_{sm}). \quad (25)$$

where d_{sm} is called the maximal step size of the traversal methods $d_i(x) \in W$. Then, we have the following correctness theorem for the algorithm of random walk strategy.

Theorem 11. *If $RW(m, n) = \langle a, b \rangle \neq \langle null, null \rangle$ then $\langle a, b \rangle$ is a Pareto front pair according to P with respect to $\|\cdot, \cdot\|$ and δ, if $d_{sm}/c^n < \delta$, where n is the number of steps.*

Proof. If $RW(m, n) = \langle a, b \rangle \neq \langle null, null \rangle$ then the condition of the If-statement in step (4) is **false**. Thus, the For-loop of Step (5) is executed. It is easy to see that the For-loop in *Step 5 Refinement* in the algorithm terminates.

Similar to the proof of Theorem 9, by the definition of d_{sm} and assumption (25), we can prove that the following is a loop invariant of the loop by induction on the number i of iterations of the loop body.

$$\|x, y\| \leq \frac{d_{sm}}{c^i} \wedge P(x) \neq P(y).$$

When the loop exits, $i = steps = n$. After executing the assignment statements $a = x$ and $b = y$, the following is true by Hoare logic.

$$\|a, b\| \leq d_{sm}/c^n \wedge P(a) \neq P(b).$$

Therefore, the theorem is true by Definition 1. □

Example 6. For example, by applying the random walk strategy on a test set containing 300 random test cases, 1000 random walks generated 805 pairs of Pareto front test cases, as shown in Fig. 6, where the walking distance was 20 steps.

In this example, the number n of steps is also 20. By the definition of $upward(x)$, $downward(x)$, $leftward(x)$ and $rightward(x)$ traversal methods, we have that $d_s = 0.2$, if the distance function $\|x, y\|$ is $Eucl(x, y)$. As in Examples 4 and 5, by the definition of $mid(x, y)$, we have that $c = 2$. By Theorem 11, the distance δ between each pair in the Pareto front satisfies the following inequality.

$$\delta \leq \frac{d_s}{c^{20}} = 0.2 \times \frac{1}{2^{20}}.$$

5. Empirical evaluation

We have conducted empirical evaluations of the proposed test strategies to determine their practical applicability for detecting borders between subdomains. In particular, we answer the following two research questions:

- *RQ1: Capability*. Are the exploratory strategies *capable* of discovering the borders between subdomains?
- *RQ2: Cost*. Are the exploratory strategies *costly* for discovering the borders between subdomains?

Capability is the probability of a test strategy returning a Pareto front pair when executed. The expected size of a Pareto front set produced by a strategy can then be calculated as $C_m \times W$ pairs, where C_m is the strategy's capability for testing classifier m and W is the number of invocations of the strategy, called the *number of walks* in the sequel.

Cost is related to the amount of computational resources needed to find a Pareto pair. We measure the cost using the average number of test executions of the classifier for discovering each Pareto pair, since the specific time and storage space depends on the classifier. Note that the strategies do not require manual labelling of the test cases or any form of test oracle. Therefore, the time taken to complete the testing process can be estimated as

$$Time = W \times C_m \times E_m \times s \quad (26)$$

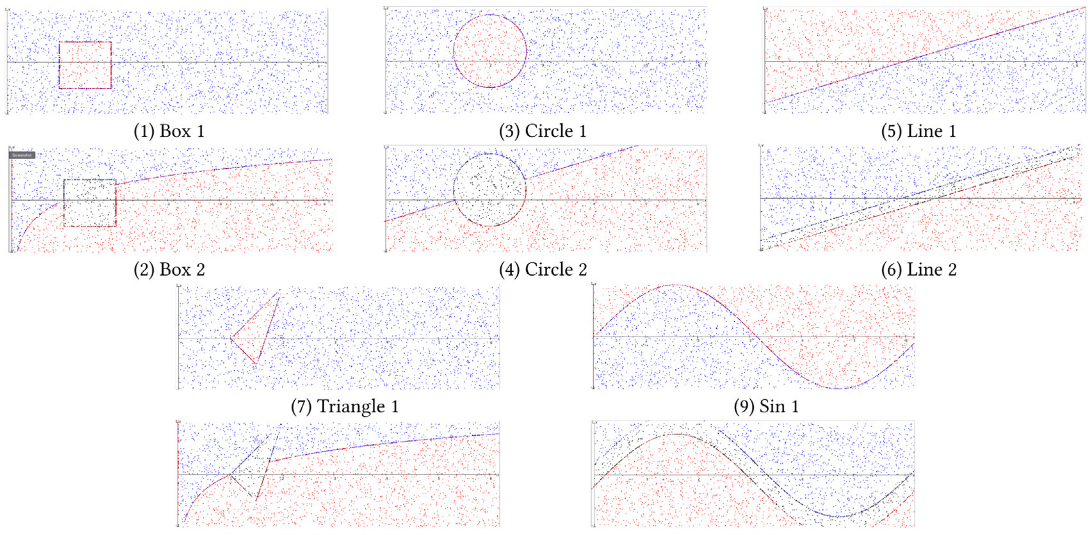


Fig. 7. Illustration of the sample applications.

where C_m and E_m denotes the strategy's capability and cost for testing the model m, and W is the number of walks and s the average time taken by each invocation of the classifier.

We have conducted two empirical evaluations of the proposed test strategies. The first is a set of controlled experiments with 10 hand-coded classifiers on two-dimensional continuous numerical features. The second is a set of case studies with 16 machine learning models built by training on three real-world datasets. Both evaluations were conducted using the automated datamorphic testing tool Morphy. The raw data collected, source code of the test systems, test scripts, etc. are all available on GitHub repository together with the executable code of the automated testing tool Morphy for download.[1] This section reports the results of these empirical studies.

5.1. Controlled experiments

5.1.1. Design and conduct of the experiments

The goal of the controlled experiments is to study the factors that affect the cost and capability of these test strategies in finding Pareto front pairs between subdomains. In doing so, we demonstrate that Pareto front pairs can represent borders between subdomains; the aim is not to compare the strategies, however.

The experiments are carried out with the ten classifiers shown in Fig. 7. These classifiers are all on the same input domain of two-dimensional real numbers in the range of $[0, 2\pi] \times [-1, 1]$. As shown in Fig. 7, they are continuous numerical feature-based classifiers.

The choice of the subjects enables us to visually display the Pareto fronts obtained from executing the test strategies so that we can verify the results against the theoretical borders between the subdomains. This has been done visually for a large number of random samples taken from the Pareto fronts and all have been found to be correct. For example, Fig. 7 shows some example screen snapshots of the visualisations of these test results. Each figure contains both the random test cases from which the starting points were selected and the test cases generated through testing. Figs. 4–6 contain only the latter.

In addition to the visual validation of the outputs of the tests, the strategies are executed repeatedly 10 times for each number of walks. The number of executions of the classifiers and the number of mutants generated were collected for statistical analysis of the capability and cost of the strategies. The following subsections reports this analysis.

5.1.2. Main results

- *Results of experiments with the directed walk strategy*

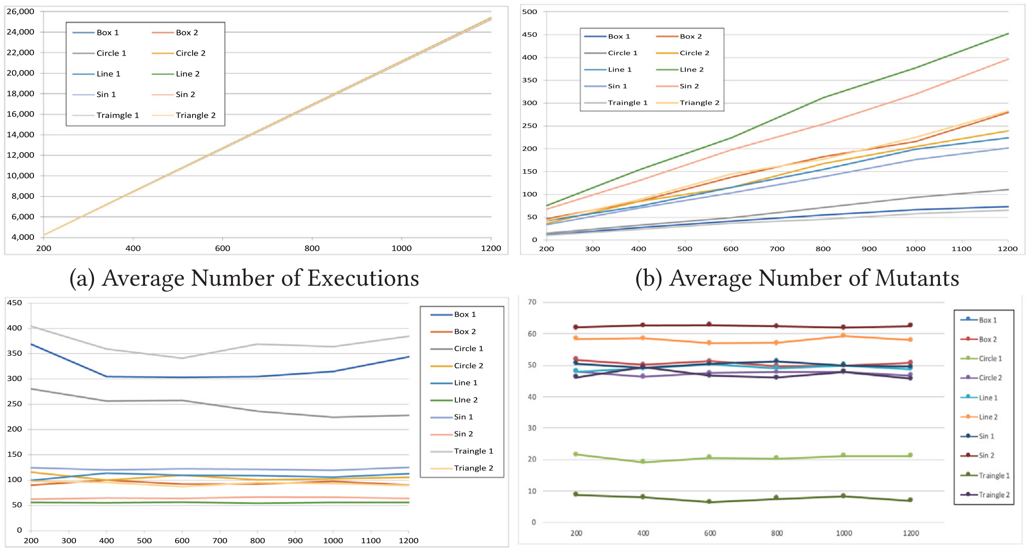
The controlled experiments on the directed walk strategy consisted of randomly selecting a number of test cases from the uniform distribution and walking 20 steps in one direction using the upward datamorphism. Both the average number of test executions of the subject program under test and the average number of mutant test cases generated (i.e. the number of Pareto front pairs) are recorded.

The experimental data shows that the number of mutant test cases generated with the directed walk strategy increases linearly with the number of walks; see Fig. 8. Similarly, the number of test executions is also linear with respect to the number of walks. In Fig. 8, the x-axis is the number of random seed test cases, which equals the number of walks, and the y-axes of (a) and (b) are the average numbers of test executions and mutant test cases, respectively. In (a), the average numbers of test executions on various subject programs are so close to each other that they are not visually separable. The y-axis of (c) measures the average cost as the number of test executions per test case in the generated Pareto front. We can see that this is fairly invariant for each subject as the former ranges from 200 to 1200. Similarly, (d) shows the average capability remains invariant when the number of walks increases.

- *Results of experiments with the random walk strategy*

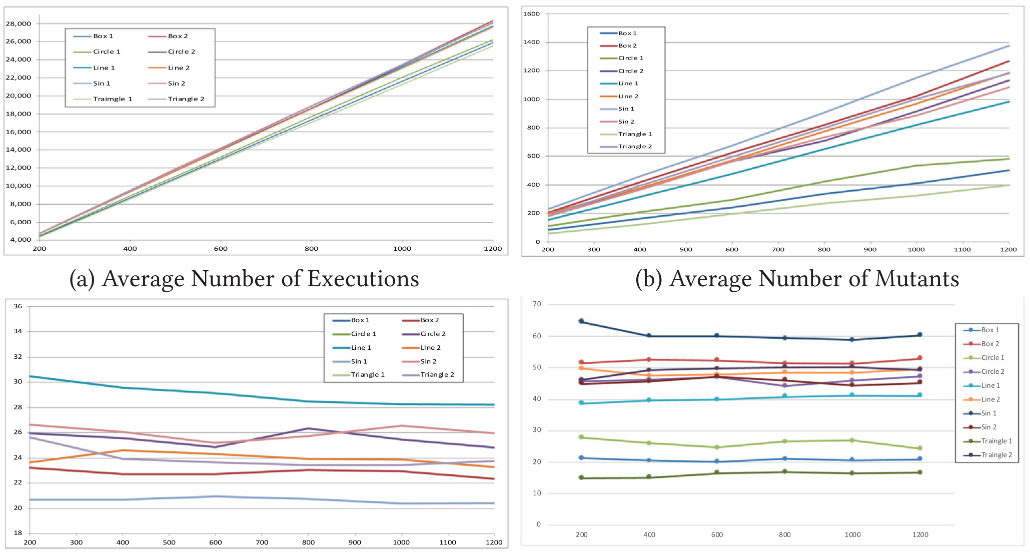
The random walk strategy is parameterised by the number of seed test cases and the number of walks starting from them. So, we fix the first parameter at 200 seeds and vary the number of walks, and then we fix the second parameter at 800 walks and vary the number of seeds. Fig. 9 shows the results of the first set of experiments with the random walk strategy. Fig. 9(a) and (b) clearly shows that the number of runs and the size of Pareto fronts increase linearly with the number of walks, while the cost and capability remains mostly invariant as shown in (c) and (d).

[1] The URL of the GitHub repository is https://github.com/hongzhu6129/ExploratoryTestAI.git.


(a) Average Number of Executions (b) Average Number of Mutants

(c) Average Cost (d) Average Capability

Fig. 8. Results of the directed walk strategy with variable number of walks.


(a) Average Number of Executions (b) Average Number of Mutants

(c) Average Cost (d) Average Capability

Fig. 9. Results of the random walk strategy with variable numbers of walks.

Similarly, Fig. 10(a) and (b) shows that the number of runs increases slightly as the number of seed test cases increases, while the size of generated Pareto front remains almost invariant. Moreover, the cost and the capability remain almost invariant as the number of seed test cases increases as shown in Fig. 10(c) and (d), respectively.

• *Results of the experiments with the random target strategy*

The random target strategy only has one parameter: the number of pairs of test cases selected at random. The experiments are conducted with this parameter ranging from 200 to 1200. The results, as shown in Fig. 11(a) and (b), are that the average number of test executions and the average size of generated Pareto front are linear in the number of walks for all subject programs. The test cost, as shown in Fig. 11(c), increases slightly with the number of walks since the average number of test executions needed to generate a test case in the Pareto front decreases as the number of walks increases. However, the capability remains invariant with the number of walks as shown in (d).

5.1.3. Discussion

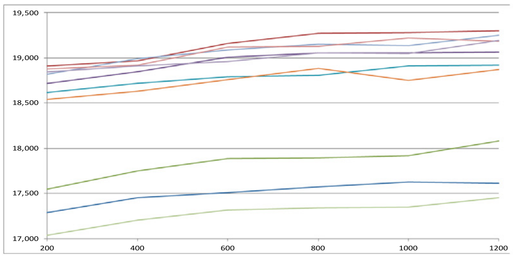
From the experiments, we observed the following phenomena in addition to the results stated above.

• *Factors influencing cost and capability*

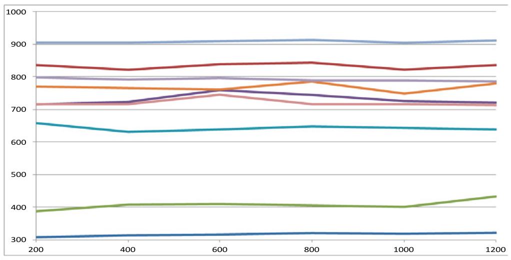
The test cost of the strategies on various subject programs are summarised in Table 1 and depicted in Fig. 12, where larger numbers indicate higher test cost.

The data show that for each strategy, the test cost and capability vary significantly according to the subject programs. However, for each strategy, test cost and capability of Box 1 are lower than Box 2, Circle 1 is lower than Circle 2, and so on. This phenomenon is not a coincidence.

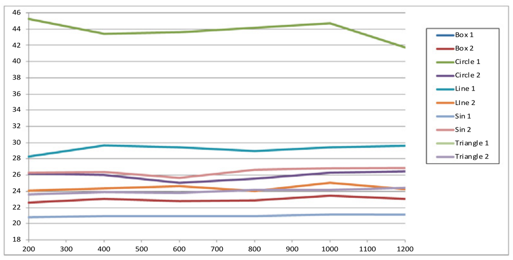
From the theorems given in Section 4, we can see that the capability for the directed walk strategy is determined by the



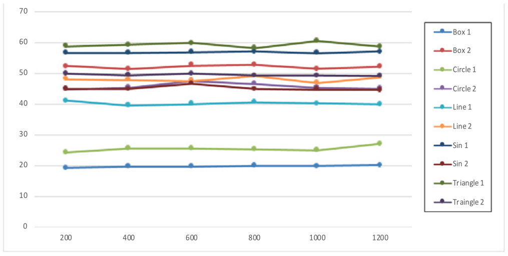
(a) Average Number of Runs



(b) Average Number of Mutants

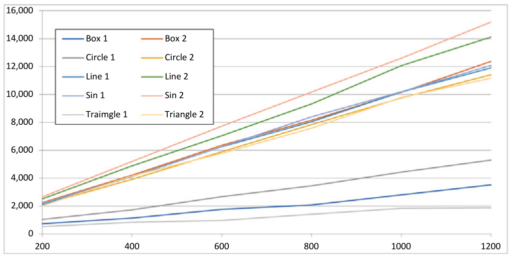


(c) Average Cost

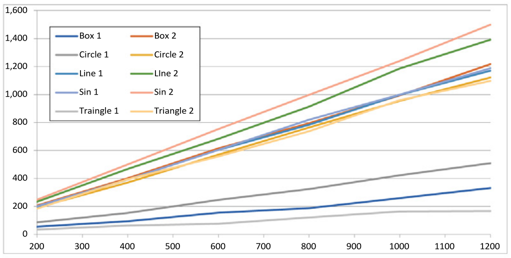


(d) Average Capability

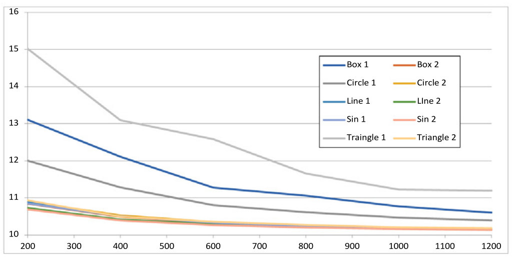
Fig. 10. Results of the random walk strategy with variable number of seeds.



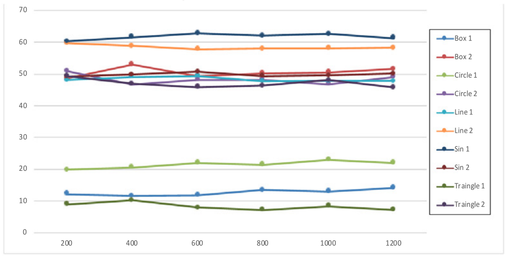
(a) Average number of runs



(b) Average number of mutants



(c) Average Cost



(d) Average Capability

Fig. 11. Results of the random target strategy.

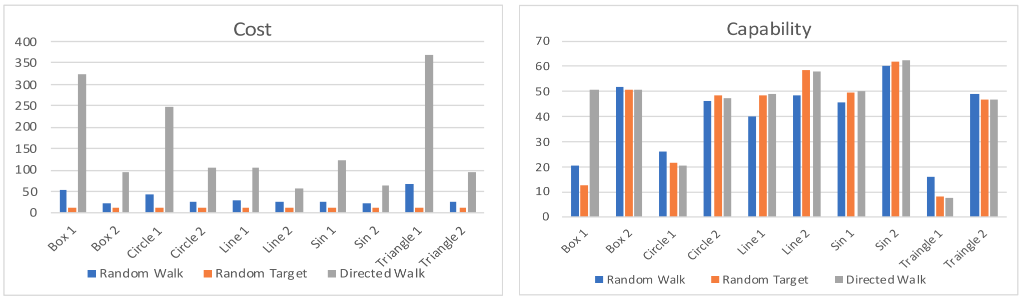


Fig. 12. Test cost and capability on subject programs.

probability that there is a border between two subdomains in the right direction from a test case and within the walking distance. For the random target strategy, it is determined by the probability that two random test cases fall in two different subdomains, and for the random walk strategy, it is determined by the probability that there is a border near to a randomly selected test case. For

Table 1
Summary of test cost and capability.

Subject	Directed walk		Random walk		Random target	
	Cost	Cap	Cost	Cap	Cost	Cap
Box 1	323.45	50.53	52.46	20.72	11.49	12.69
Box 2	93.85	50.53	22.83	51.59	10.38	50.53
Circle 1	247.32	20.67	42.59	26.03	10.93	21.49
Circle 2	105.82	47.32	25.50	46.01	10.41	48.31
Line 1	105.82	49.15	29.02	40.13	10.41	48.25
Line 2	55.76	58.03	23.94	48.56	10.33	58.40
Sin 1	122.35	50.10	20.65	45.51	10.38	49.76
Sin 2	64.75	62.34	26.03	60.54	10.31	61.76
Triangle 1	370.38	7.62	66.79	16.06	12.46	8.33
Triangle 2	93.19	46.96	23.98	49.08	10.41	47.01
Avg	**158.27**	**44.32**	**33.38**	**40.46**	**10.75**	**40.65**

test cost, the more Pareto front pairs found, the more runs of the classifier will be required to refine the pairs of test cases in order to reduce the distance between each pair.

Two implications follow from these properties. First of all, given a classification application, one should select the most cost efficient strategy to explore the Pareto fronts between subdomains based on the understanding of the application. The data obtained from our experiments are not sufficient to compare the strategies on their cost. This is because the probability of finding a pair in the Pareto front heavily depends on the size and location of the subdomains of the classification application. Our subjects in the experiments may not be representative of the distribution of the parameters in real applications. Secondly, we now have an explanation why the number of pairs generated for the Pareto front is a linear function of the number of walks since the results of a walk is independent of the results of its predecessors.

Moreover, although the cost is mostly determined by the size, shape and location of the subdomains that the program classifies, for directed walk and random walk strategies, it is also affected by the number of steps walked and the number of iterations in the refinement. The number of steps walked influences the probability of finding two points in different subdomains and also the total number of test executions. The longer the walk, the more likely one is to find two points in different subdomains, but this requires more test executions. Thus, a balance between these two contradictory factors of cost must be made to achieve the best test effectiveness.

Finally, the number of iterations in the refinement loop controls the distance between the pairs of test cases in the Pareto fronts generated. It has no impact on capability, i.e. the probability of finding two data points in different subdomains, but it does have an affect on test cost. The shorter distance requires more iterations, and thus more test executions, and therefore, it is more costly. For random walk and directed walk strategies, the number of iterations can be selected according to the formula given in the correctness theorems given in Section 4. For the random target strategy, usually more iterations are required than the other two strategies.

- *Validity of the experiments*

As pointed out at the beginning of the section, the experiments are designed to determine which factors have an effect on the capability and cost of the strategies. The subject programs used in the controlled experiments are manually coded by the authors. They have been designed in such a way that their subdomains are of typical shapes in data mining and machine learning applications (Aggarwal, 2015; Mohri et al., 2012; Shalev-Shwartz and Ben-David, 2014). As discussed above, they provide insight into the factors that affect capability and cost.

The manual examinations of the Pareto fronts generated by the test strategies confirmed that they are indeed test cases very close to the borders of subdomains. The phenomena observed from the experiments are consistent with the predictions made from the theorems. However, the specific data about cost and capability obtained from the experiments depends on the specific features of the subdomains such as their sizes and locations. Therefore, the experiment data do not answer the question whether the test strategies are applicable to testing real machine learning applications. This issue is addressed in the case studies reported in the next subsection.

5.2. Case studies

This subsection reports a set of case studies with the exploratory testing of machine learning and data analytics applications using the test strategies.

5.2.1. Design and conduct of the case studies

The procedure for the case study consists of the following steps:

1. Select sample applications of classifiers.
2. For each selected sample,
   (a) Download the dataset.
   (b) Construct classifiers by applying machine learning techniques on the dataset.
   (c) Develop test system according to the specification of the test systems defined in Section 3.
   (d) Write test scripts in Morphy's test scripting language for repeated executions of the experiments and collection of data.
   (e) Execute test strategies on the test classifiers by running the test scripts.

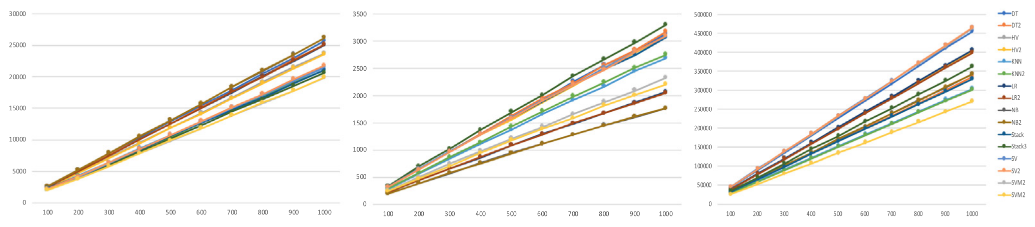
The following describes each step in detail.

- *Sample datasets*

The following three datasets were selected at random from the well-known Kaggle collection of datasets for machine learning and data analytics applications. They were as follows:

(1) *Red Wine Quality*. This dataset concerns red varieties of the Portuguese "Vinho Verde" wine (Cortez et al., 2009). There are 11 physicochemical variables as inputs (i.e. there is no data about grape types, wine brand, wine selling price, etc.) and the output is a classification of wine quality as a number from 1 to 10. The classes are ordered but not balanced in that there are many more normal wines than excellent or poor ones.

(2) *Mushroom Edibility*. This dataset concerns hypothetical samples of 23 species of gilled mushrooms in the Agaricus and Lepiota family drawn from The Audubon Society Field Guide to North American Mushrooms (Society, 1981). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one in the dataset. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom, i.e. no rule like "leaflets three, let it be" for poison oak and poison ivy. The dataset has been available to researchers on data mining and machine learning for 30 years.



(a) Random Walk on Red Wine (b) Random Target on Mushroom (c) Directed Walk on Bank

Fig. 13. Variation of the number of runs with the number of walks.

Table 2
Summary of datasets.

Dataset	Records	Classes	DF	NF	CF	Features
Red wine quality	1599	8	0	0	11	11
Mushroom edibility	8124	2	22	0	0	22
Bank churners	10 127	2	5	11	3	19

(3) *Bank Churners.* This dataset concerns credit card customers and can be used to predict churners, who are bank customers who leave the credit card service. It consists of more than 10,000 real data items with 19 features about customer's age, salary, marital status, credit card limit, credit card category, etc. It is, however, considered to be a difficult task to train a model to predict churning customers.

All three datasets are available from the Kaggle repository.[2][3][4] The first two datasets are commonly used in research on machine learning and data analytics to determine which physiochemical properties make a wine good and which features are most indicative of a poisonous mushroom, respectively. As well as Kaggle, they can also be found at the UCI machine learning repository.[5][6] The Bank Churners dataset originates from a LEAPS website,[7] which specialises in application of data analytics and machine learning techniques to solve business problems.

Table 2 summarises the datasets used in the case study. The column *Records* gives the number of records in the dataset and *Classes* is the number of classes (subdomains) in the classification. Columns *DF*, *NF* and *CF* are the numbers of discrete non-numerical features, discrete numerical features and continuous numerical features, respectively. The column *Features* shows the total number of features. We can see that the dataset *Red Wine Quality* is a continuous numerical data space, whereas the dataset *Mushroom Edibility* is a discrete non-numerical data space, and the *Bank Churners* dataset is a hybrid data space.

- *Construction of machine learning models*

Since the goal of the case study is to demonstrate that our test strategies are applicable to real machine learning applications, we have used the datasets to train models that use a wide variety of machine learning techniques. This enables us to demonstrate that our testing techniques are effective on both low-quality and high-quality models as well as on different types of models.

The training consists of executing a Python program, adapted from code posted on the Kaggle website and selected at random again. For each dataset, we build 16 different models, as shown in Table 3. The Python programs for training and invoking the models as well as all datasets used in the case study can be found on the project's GitHub repository; see Footnote 1 for the URL.

A total of 48 models were constructed. Their accuracy varies from 49.9% to 100%; see Appendix B.1 for details. It is worth noting that no effort was spent to construct a model of high quality because the purpose of the experiment is to determine if the strategies are capable and cost efficient for models of all different kinds of quality.

- *Development of test systems*

The test system for the Red Wine Quality dataset was a straightforward implementation of the appropriate algorithm in Section 3 and the code for the experiments was a clone of the code written for Section 5.1. The main difference in the test system is that the executions are performed by invoking programs in Python through executing test morphisms in Java.

The test system for Mushroom Edibility was made by refactoring the test system for Red Wine Quality to make the code common to both ready for reuse. Once again, the datamorphisms were a straightforward implementation of the definitions in Section 3. Similarly, the test system for Bank Churners prediction is again a straightforward implementation of the algorithms given in Section 3.

- *Executions of test strategies*

As with the controlled experiments in Section 5.1, the test strategies are applied to each classifier to generate the Pareto fronts and the same kinds of data are collected from their executions.

In particular, both the random target and random walk test strategies were executed with varying numbers of walks (10 times in each case) ranging from 100 to 1000 in order to calculate the average number of mutant test cases generated, i.e. the number of test cases in the generated Pareto front. The directed walk strategy was executed with starting points of 100, 200, ..., 1000 test cases selected at random from the original dataset on all directions (i.e. each unary datamorphism) for 10 times; the average of these directions was calculated for each of the models.

The repeated executions of the test strategies were conducted by invoking test scripts written in Morphy's test scripting facility. The test scripts can be found in the GitHub repository.

5.2.2. Main results

- *Numbers of runs and mutants*

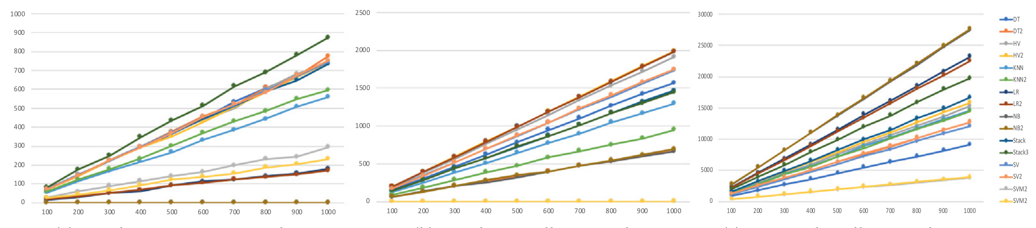
The case studies clearly show that for all machine learning models, the average numbers of runs of the model increase linearly with the number of walks made when executing the test strategies. Fig. 13 shows some typical examples.

Similarly, the average numbers of mutant test cases (i.e. the points in Pareto fronts generated by strategies) increase linearly

[2] https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009.
[3] https://www.kaggle.com/uciml/mushroom-classification.
[4] https://www.kaggle.com/sakshigoyal7/credit-card-customers.
[5] https://archive.ics.uci.edu/ml/datasets/wine+quality.
[6] https://archive.ics.uci.edu/ml/datasets/Mushroom.
[7] https://leaps.analyttica.com/home.

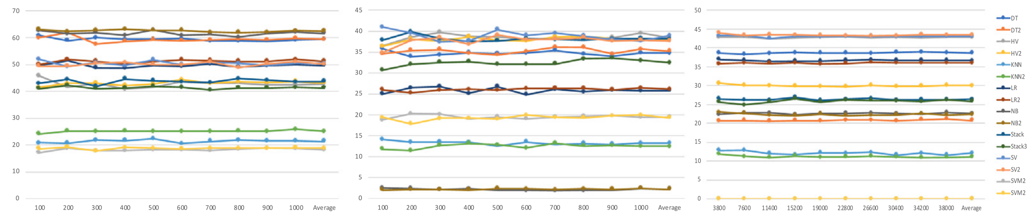
Table 3
Machine learning models constructed for each dataset.

Name	Type	Details
LR	Logistic Regression	Trained on whole dataset
LR2	Logistic Regression	Used train-test 90-10 split
KNN	K-Nearest Neighbours	Trained on whole dataset
KNN2	K-Nearest Neighbours	Used train-test 90-10 split
DT	Decision Tree	Trained on whole dataset
DT2	Decision Tree	Used train-test 90-10 split
NB	Naive Bayes	Trained on whole dataset
NB2	Naive Bayes	Used train-test 90-10 split
SVM	Support vector machine	Trained on whole dataset
SVM2	Support vector machine	Used train-test 90-10 split
SV	Ensemble via Soft voting	Trained on whole dataset; LR+KNN+DT
SV2	Ensemble via Soft Voting	Used train-test 90-10 split; LR+KNN+DT
HV	Ensemble via Hard Voting	Trained on whole dataset; LR+KNN+DT
HV2	Ensemble via Hard Voting	Used train-test 90-10 split; LR+KNN+DT
Stack1	Ensemble via Stacking	Used train-test 90-10 split; KNN as Meta; LR2+KNN2+DT2+HV2
Stack3	Ensemble via Stacking	Used train-test 90-10 split; LR as Meta; KNN2+DT+SV2+HV2



(a) Random Target on Mushroom (b) Random Walk on Bank (c) Directed Walk on Red Wine

Fig. 14. Variation of the number of mutants with the number of walks.



(a) Random Target on Red Wine (b) Random Walk on Mushroom (c) Directed Walk on Bank

Fig. 15. Variation of capability with the number of walks.

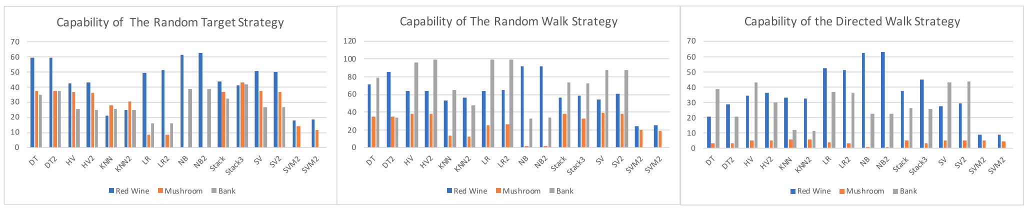


Fig. 16. Capabilities of testing different ML models.

with the number of walks from 100 to 1000. Again, this is for all machine learning models. Three typical examples are shown in Fig. 14.

The data of the case studies confirmed the observations made in the controlled experiments.

- *Capability of discovering borders*

The capability of each test strategy in discovering border points for each machine model, measured as the probability of finding a border point via a walk, remains invariant in the number of walks as shown in Fig. 15. However, the capability varies significantly over different machine learning models; see Fig. 16.

- *Cost*

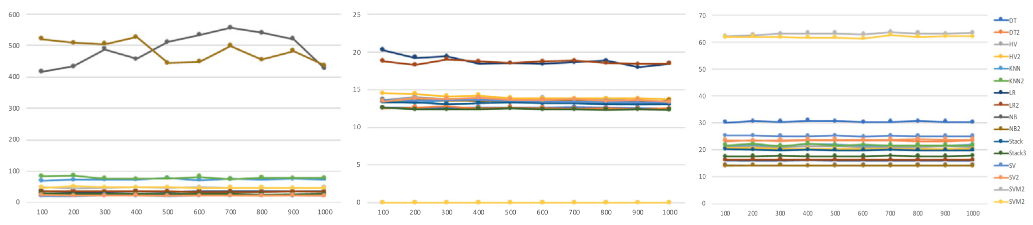
As was seen with the controlled experiments in Section 5.1, the case studies show that the cost of the strategies was mostly invariant as the number of walks increases; see Fig. 17 for some typical examples. The cost for each model is shown in Fig. 18.

5.2.3. *Discussion*

- *Answers to the research questions.*

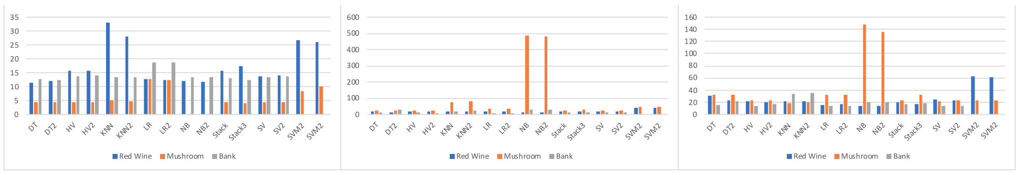
From the data collected from the case studies, we can draw the following conclusions.

First, the data of the case study are consistent with the observations made in the controlled experiments that both capability



(a) Random Walk on Mushroom (b) Random Target on Bank (c) Directed Walk on Red Wine

Fig. 17. Variations of cost with numbers of walks.



(a) Random Target (b) Random Walk (c) Directed Walk

Fig. 18. Cost of testing different ML models.

Table 4
Summary of the capability and cost of the strategies.

Strategy	Subject	Cost				Capability			
		Max	Min	Avg	StDev	Max	Min	Avg	StDev
Directed Walk	Red Wine Quality	63.03	14.12	25.70	0.15	62.89	8.79	35.74	0.24
	Mushroom Edibility	32.63	18.90	25.57	0.38	5.79	0.80	4.10	0.06
	Bank Churners	35.56	14.07	19.26	0.21	43.43	0.00	25.75	0.21
Random Target	Red Wine Quality	33.14	11.47	17.39	0.46	62.51	18.18	43.62	0.72
	Mushroom Edibility	12.61	3.92	6.23	0.26	43.05	0.00	25.18	0.59
	Bank Churners	18.81	12.40	14.06	0.18	41.66	0.00	25.60	0.64
Random Walk	Red Wine Quality	40.87	14.31	20.71	0.39	91.87	24.12	61.61	0.87
	Mushroom Edibility	488.50	21.42	92.01	6.35	38.87	2.15	25.87	0.63
	Bank Churners	30.34	8.10	15.94	0.28	99.43	0.00	62.83	0.47

and cost of the strategies heavily depends on the model under test, but is invariant in the number of walks. In other words, both cost and capability are constants that only vary with the model under test.

Second, the strategies are capable of discovering borders between subdomains. The overall average of the capabilities of all three strategies is 34.48%. The average capabilities of the directed walk, random target and random walk strategies over three subjects are 21.86%, 31.47% and 50.10%, respectively. The highest capability reached was 62.83% in testing bank churner prediction using the random walk strategy. The average capabilities are almost all above 25% except that the average capability of testing mushroom edibility models using the directed walk strategy is only 4.10%. Table 4 shows the maximal, minimal, and the average capability and cost of the test strategies over different models.[8]

Third, the case study also clearly demonstrated that applying exploratory strategies is cost efficient for discovering borders between classes; also see Table 4. The overall average cost of three strategies over all subjects is 26.32, which means that on average one would detect a border point by executing the machine learning model on about 27 test cases. In other words, within a fraction of second, a large number of border points can be found by applying these exploratory test strategies. The best cost efficiency was achieved in the testing of mushroom edibility models using the random target strategy, where the average cost over 16 models is 6.23. In contrast, the worst cost of 92.01 is observed also when testing mushroom edibility but using the random walk strategy.

Fourth, comparing with the data of the controlled experiments, we observed that the costs and capabilities of the strategies in the case study are compatible to those of controlled experiments, although the dimensions of the input data spaces of the real-world examples are significantly larger than those coded classifiers. This indicates that the approach is scalable to high dimensional data spaces.

Moreover, the data of the case study provides some useful hint for the choice of strategies when testing a machine learning application. The data show that on average, the random walk strategy is the most capable in detecting borders. However, the walk may require many steps to find a border point. Thus, it could be slightly less cost efficient than the random target strategy in many cases. For the directed walk strategy, searching for borders in all directions is very much like a brute force search. Thus, it could be of higher cost in general.

Finally, in the case study, we observed a few cases where exploratory strategies performed poorly. These cases provide some insight for how to choose from the proposed strategies.

Among the worst capabilities observed in the case studies is that of the directed walk strategy which performed poorly on testing mushroom edibility with an average capability of 4.10% over 16 models. The reason why directed walk performed poorly on testing mushroom edibility models is as follows.

The theorems proved in Section 4 imply that the capability of the directed walk in a given direction depends on the existence of a border in the direction from the randomly selected starting test case. If a border point is found, it only differs from the starting test case in one feature. This is a limitation of the capability of

[8] When no Pareto Front is found, the cost is infinite. In such cases, the numbers given in Table 4 for the maximal, minimal and average cost have been calculated by excluding the infinite.

the strategy. This is the case for testing the mushroom edibility, where it is rare that changing just one feature of a mushroom variety will change its edibility; usually at least two features must change.

It was also observed that the random target strategy has zero capability when used for testing the NB and NB2 models of mushroom edibility, as does all three strategies when testing the SVM and SVM2 models of bank churners. The reason for the poor performances is as follows.

The random target strategy discovers a border point when the two starting points are in different classes. If a subdomain is small, the probability of selecting a point inside it is correspondingly small. In the extreme case, when all test cases are in the same class, no border will be discovered. The NB and NB2 models of mushroom edibility classify all mushrooms in the training dataset as poisonous. Similarly, the SVM and SVM2 models of bank churners classify all credit card customers to be non-churners so no Pareto front can be discovered by any strategy.

It is worth noting that the NB and NB2 models have the worst accuracy among all models of mushroom edibility, and SVM/SVM2 models are the worst on accuracy among the models of bank churners. They are underfit models, which means they are insufficient for classifying the input data space. Therefore, exploratory testing cannot detect the borders between subdomains.

On average, the random walk strategy achieved the best performance on capability. It can discover a border point even if all start points are in the same class; it is only required that a border exists within walking distance from the starting point. Moreover, the Pareto front found may be different from the starting point on many features. Although its cost is not the lowest of the three, it balances capability and cost best of the three.

- *Length of Execution Time.*

The real cost of the testing strategies in terms of the lengths of execution time required to generate a Pareto front for a classifier depends on the speed of the computer system, the time needed to invoke the classifier to classify an input data, and the number of walks to be executed. The measure of test cost in terms of the number of invocations of the classifier under test per pair of points in the Pareto front gives an abstract metric, which is independent of these factors while the real cost can be calculated with these factors as parameters by using Eq. (26). To give an indication to the scale of real cost, we have run each strategy 10 times for each classifier, and each time we have executed 1000 walks and recorded the clock times spent and the sizes of Pareto fronts generated. The testing tool Morphy was run on a Windows PC with Intel Xeon x64 CPU E3-1230V5 3.40 GHz and 32 GB memory.

Table 5 shows the average numbers of Pareto front pairs generated per second for various coded classifiers used in the controlled experiments. From these data, the average real cost of generating Pareto fronts of a certain size, such as 1000 pairs of points, can be easily calculated by the formula $Time = \frac{P}{RC}$, where P is the size of Pareto front, RC is the data of real cost given in Table 5, i.e. the average number of Pareto front pairs per second.

The data shows that, for coded classifiers, on average, generating a Pareto front consisting of 1000 pairs of points only took less than 0.4 s. The worst case, for directed walk strategy, for the same size of Pareto front was 1.66 s and the best case, for random target strategy, took 0.27 s.

Table 6 shows the results of testing those real ML models used in our case studies. It gives the average number of pairs generated per second for various types of ML models using different exploratory strategies in the same experimental setup, where DW, RT and RW stand for Directed Walk, Random Target and Random Walk strategies, respectively.

Table 5
Average number of pairs generated per second for coded classifiers.

Classifier	Directed walk	Random target	Random walk	Average
Box 1	645.93	3059.24	1919.92	1875.03
Box 2	2734.57	3532.37	2954.01	3073.65
Circle 1	1084.63	3730.02	2281.91	2365.52
Circle 2	2956.22	3591.11	2909.88	3152.40
Line 1	2421.86	3709.12	2749.14	2960.04
Line 2	2434.26	3610.46	2985.71	3010.14
Sin 1	2133.10	3733.23	2880.03	2915.45
Sin 2	2500.64	3653.87	3090.02	3081.51
Triangle 1	601.53	3104.84	1853.88	1853.42
Triangle 2	2773.01	3697.53	2932.50	3134.35
Average	2028.57	3542.18	2655.70	2742.15

The data shows that it took less than 10 s to generate Pareto fronts of 1000 pairs. In the worst case, which is when testing the Stack model of Mushroom Edibility using the directed walk strategy, it took an average of 117.11 s (less than 2 min) in 10 executions of the test strategies to generate 1000 pairs. The best case is when testing the Logistic Regression model LR of Bank Churners using the random walk strategy. This took less than 2 s to generate the same number of pairs.

There are two machine learning models in our case study that do not have any border between classes as our exploratory testing discovered. They are the naïve Bayes model NB of mushroom edibility and the Support Vector Machine model SVM of bank churners prediction. Table 7 shows the average lengths of time that the strategies completed the search for borders by taking 1000 walks to test these two models. In the worst case, it took less than 2 min, while in most cases it took between a fraction of a second and a few seconds.

In general, the time taken to execute a test strategy heavily depends on how fast the classifier under test is for classifying an input data. Our experiment data presented in the previous sections shows that the time to execute the strategies increases linearly with the number of walks and with the number of pairs of border points generated. Therefore, the experiment data with real machine learning models indicate that to generate a Pareto front containing 1000s of pairs, on average we only need 10s of seconds. It is highly efficient for practical uses of the strategies.

- *Validity of the Conclusions.*

The case studies have been conducted on datasets selected at random from a large library with each dataset representing a different type of classifier system. It is possible that the datasets chosen had special properties that had an impact of the results but this threat to validity can be eliminated by repeating the case studies on other datasets.

The case studies used a wide range of models of different types and of different quality (e.g. of different accuracy). They were constructed by using Python code selected from the Kaggle website at random. The distribution of the quality among these models may be not representative of the models in a real production environment. Thus, the statistics may be biased. However, due to the lack of data on the distributions of model quality, we are unable to eliminate such a potential bias. The way to improve this aspect is to use the test strategy in a real production environment.

The test systems were implemented by the authors according to the formal definitions given in Section 3. They were debugged and tested on a large number of test cases. A threat to the validity of the case study is the existence of bugs in the test system, which may have impact on the correctness of the data. The source code of the test systems is written in Java and freely available from

Table 6
Average number of pairs generated per second for real ML models.

ML	Red wine quality			Mushroom edibility			Bank churners			Average
model	DW	RT	RW	DW	RT	RW	DW	RT	RW	
DT	84.46	266.68	203.21	131.52	329.42	176.43	278.04	188.77	327.33	220.65
HV	21.68	32.13	28.62	22.56	119.62	23.43	33.68	35.09	47.05	40.43
KNN	30.16	21.95	37.70	36.73	140.34	9.23	19.68	47.57	39.35	42.52
LR	272.68	260.63	202.64	135.44	307.29	121.53	276.27	204.84	526.73	256.45
NB	146.81	156.79	148.69	–	–	–	166.61	231.19	113.20	160.55
Stack	9.03	13.21	10.86	8.54	46.29	8.76	10.66	13.83	14.53	15.08
SV	8.62	20.31	14.39	11.52	63.03	12.09	17.87	18.92	22.13	20.99
SVM	37.02	68.94	52.74	120.37	306.60	61.98	107.94	–	–	–
Avg	76.31	105.08	87.36	66.67	187.51	59.06	114.69	105.74	155.76	105.70

Table 7
Lengths of time (second) to complete search when no border in the classifier.

ML model	Directed walk	Random target	Random walk
NB-Mushroom	14.83	0.25	2.87
SVM-Bank	102.47	0.69	7.51

Table 8
Comparison of related testing methods.

Test method	Test design	Execution	Learning	Steering
Proposed method (ET)	✓	✓	✓	✓
Fuzz testing	✗	✓	✗	✗
Data mutation testing	✓	✓	✗	✗
Adaptive random test	✗	✗	✗	✓
Metamorphic testing	✗/✓	✓	✗	✓
Search-based testing	✗/✓	✗/✓	✗	✓
Domain testing	✗	✗	✗	✗

GitHub for inspection. We are reasonably confident that the test system has no serious bugs.

The process of the case studies is highly automated by executing test scripts written in Morphy's test scripting language. Manual operational errors in the conduct of the case studies can be eliminated to the highest extent. However, there may be bugs in the test script and in the Morphy testing tool. Such bugs form a threat to the validity of the conclusions drawn from the data. We believe that this threat should have a minimal impact, however, as the Morphy tool and test scripts have been tested, too. Morphy is available for download and use for free. The test scripts are also available on GitHub for download and inspection. The whole case study can be repeated easily.

Finally, the observations made in the case studies and the conclusions drawn from the data are consistent with the observations made in the controlled experiments and what the formally proved theorems imply from the formal definitions and the algorithms. Therefore, we can confidently conclude that the conclusions drawn from the cases studies are valid and can be generalised to other machine learning models built via supervised training on datasets.

6. Related work

The most closely related work is exploratory testing (ET). We will review the current state of research in this field and summarise our contributions to it. We will also discuss the similarities and differences between our work and adaptive random testing (ART), fuzz and data mutation testing, metamorphic testing (MT), and search-based testing (SBT). Finally, since the work of this paper is partially inspired by the traditional testing method of domain testing, we will also briefly discuss the applicability of that method to machine learning models.

6.1. Exploratory testing

ET has been widely applied to many types of software systems, but most successfully to GUI-based systems; see, for example, Whittaker (2009). Pfahl et al. (2014) reported an online survey of Estonian and Finnish software developers and testers on their uses of exploratory testing in practice, revealing that a majority used it intensively for usability-critical, performance-critical, security-critical and safety-critical software. However, as far as we know, there is no report on the systematic application of ET for testing AI applications.

Research on ET exists that evaluates its fault detection effectiveness and efficiency, including reports on its effectiveness in practice. Itkonen and Rautiainen (2005), Itkonen et al. (2007) and Itkonen and Mäntylä (2014) were amongst the first. They used students as subjects to compare ET with traditional software testing techniques that based on pre-designed test cases (TCT). Through replicated experiments, they found that ET had the same effectiveness in fault detection but greater time-efficiency because less design effort was needed. Moreover, TCT produces more false-positive defect reports than ET.

Afzal et al. (2014) conducted a controlled experiment with 24 practitioners and 46 students who performed manual functional testing to compare the effectiveness of exploratory testing against traditional test techniques. Unlike Iktonen et al. they reported that ET found significantly more defects, including those at varying levels of difficulty, type and severity. Also unlike Iktonen et al. they did not report that ET reduced the number of false-positive defect reports.

However, both of these experiments were conducted on traditional software and since AI applications have different failure modes and faults, it is unclear whether ET is effective and efficient for machine learning applications.

Since ET is used as a manual testing method, research on it has mostly focused on the human factors that alter effectiveness and efficiency. An industrial case study by Gebizli and Sozer (2017) with 19 practitioners of different educational backgrounds and experience levels show that both factors affects efficiency but only experience affects the number of critical failures detected. Micallef et al. (2016) found that trained testers employed different types of exploratory strategies than untrained testers. The trained testers were more effective at finding input validation errors, while untrained testers tended to uncover mostly content bugs. The two groups were however equally effective at detecting logical bugs or functional UI bugs.

Shoaib et al. (2009) found that people with extrovert personalities are more likely to be good exploratory testers. Itkonen et al. (2013) found that exploratory testers applied their knowledge for test design and failure recognition differently. Martensson et al. (2021), after interviewing testers in six companies, identified nine key factors that determine the effectiveness of exploratory testing in an organisation and proposed a simple model for improving it.

On the automation of ET, Eidenbenz et al. (2016) employed artificial intelligence techniques to predict test cases that are likely to cause failure in testing an industry control software. Makondo et al. (2016) used neural networks to train test oracles to help the analysis of test results. Research has also been reported on the development of test environments to support exploratory testing. For example, ARME enables the automatic refinement of system models based on recorded testing activities of test engineers (Gebizli and Sözer, 2016). Tapir supports team collaboration in exploratory testing and reconstruction of system models (Bures et al., 2018). However, as far as we know, there is no work in the literature that automates exploratory strategies as we have done, even though many exploratory strategies have been documented in the literature such as those by Whittaker (2009) and Hendrickson (2013).

Our main contributions to exploratory testing are to apply it to machine learning applications and to automate it. By identifying the discovery of boundary values as a specific goal of testing, we demonstrated how the elements of ET can be formalised and implemented in the datamorphic testing framework to achieve test automation. Our approach can be summarised as follows.

Firstly, test design is formalised and implemented by a set of datamorphisms. Together with a test executor test morphism, these datamorphisms form a test system for exploratory testing. We introduced the notion of complete exploratory test systems, developed a systematic way to construct exploratory test systems for feature-based classifiers, and proved that such exploratory test systems are complete so that they ensure the whole data space of the model can be explored.

Secondly, steering strategies are formalised and implemented as algorithms that invoke the datamorphisms and the test executor. We then formally proved that the strategies are correct; that is, they always terminate and produce Pareto fronts that represent the borders between classes. In other words, these strategies always achieve the goal of ET: to discover the borders between classes defined by the machine learning model under test.

Finally, the strategies have been implemented in the automated datamorphic testing tool Morphy (Zhu et al., 2020, 2019a). We have also conducted empirical evaluations of the strategies to determine their capability and cost through controlled experiments and case studies. The results demonstrated that the approach can discover the borders between classes in a cost efficient way. The data also provide insight into the factors that affect capability and cost for each test strategy. This can be used to select appropriate parameters for appropriate strategies for each classification application.

6.2. Random and adaptive random testing

Generally speaking, random testing (RT) is a software testing method that selects or generates test cases through random sampling over the input domain or a profile of the software under test (SUT) according to a given probability distribution (Hamlet, 2002). As discussed in Zhu et al. (1997), RT techniques can be classified into two types: representative and non-representative.

The representative type uses the probabilistic distribution on the input domain as the input distribution for the SUT. One approach is to sample at random the operation profile of the software under test (Myers et al., 2011). Another approach is to develop a Markov model of the human computer interaction process and use it to generate random test cases (Whittaker and Thomason, 1994). Although representative RT works well for fault detection in simulation-based experiments (Duran and Ntafos, 1984; Hamlet and Taylor, 1990; Tsoukalas et al., 1993; Ntafos, 1998), its most compelling advantage is that test results naturally lead to an estimate of software reliability. However, such random testing requires a much larger number of test cases to achieve the same level of fault detection ability in comparison to test methods where the test cases are purposely designed.

In contrast, the non-representative type of RT uses a distribution unconnected to the operation of the software. The major subtype of ART methods, for example, spread test cases evenly over the entire input space (Chen et al., 2001b, 2004b, 2007, 2010) and experiments show that they improve both fault detection ability (Chen and Kuo, 2007) and reliability (Liu and Zhu, 2008). Even spread over the input space can be achieved by manipulating randomly generated test cases. Many such manipulation algorithms (called "strategies" in the literature) have been developed and evaluated, including mirror (Chen et al., 2004a), balance (Chen et al., 2006), distance (Huang et al., 2020), filter (Chan et al., 2005), lattice (Mayer, 2005), partition (Mao et al., 2020), etc. In a recent comprehensive survey of ART, Huang et al. (2019) classified these techniques into Select-Test-From-Candidates Strategies, Partitioning-Based Strategies, Test-Profile-Based Strategies, Quasi-Random Strategies, and their combinations (called Hybrid-Based Strategies). Even spread can also be achieved with evolutionary computing algorithms, as discussed later in the subsection on search-based testing.

A common feature of these ART algorithms for test case generation is that the new test cases are generated or selected based on the positions of existing test cases in the input space. This is similar to the so-called steering feature of exploratory testing. However, none of them uses the information revealed in test executions. In fact, the generation and/or selection of new test cases in ART strategies do not require the execution of the software under test at all. Of course, the most fundamental difference between ART and ET is that ART does not aim to discover the system's behaviour although evenly spreading the test cases may help indirectly. Another difference is in test design, which is the selection of a probability distribution on the input domain for ART and a decision on how to change the test cases for ET.

6.3. Fuzz and data mutation testing

Datamorphic testing evolved from data mutation testing (DMT) (Shan and Zhu, 2009) and its integration with metamorphic testing (Zhu, 2015). Data mutation testing was proposed by Shan and Zhu (2006) to generate realistic test cases that are structurally complex, such as those for software modelling tools. The basic idea is to develop a set of operators that transform existing test cases (called *seed test cases*) to new test cases (called *mutant test cases*). These operators were originally called data mutation operators, but were renamed as *datamorphisms* in Zhu et al. (2019b). Shan and Zhu (2009) also proposed that data mutation operators (i.e. datamorphisms) can indicate the correctness of the program on mutant test cases. Metamorphic relations associated with data mutation operators were formally defined in Zhu (2015) as *mutational metamorphic relations*, and called *metamorphisms* in datamorphic testing (Zhu et al., 2019b). The uses of seed makers, datamorphisms and metamorphisms in one general purpose testing tool to achieve test automation was first reported in Zhu (2015).

Data mutation testing has similarity to *fuzz testing*; see, for example, Sutton et al. (2007). However, mutation testing emphasises an engineering process of developing data mutation operators that can be used to generate meaningful and realistic test cases for the software under test, while fuzz testing randomly makes a change without first determining whether the mutants are meaningful and realistic or not. A datamorphic test system can include either random or purposeful datamorphisms or even a combination of both. Most importantly, datamorphic testing recognises other types of test morphisms and uses them to achieve test automation at a high level of strategy and test process (Zhu et al., 2019b, 2020).

6.4. Metamorphic testing

Metamorphic testing was proposed by Chen et al. (1998b) to use metamorphic relations to check test results and to generate test cases. A metamorphic relation is a relation on inputs and outputs of multiple test cases. Theoretically speaking, metamorphic relations are axioms about the software under test presented in a special form as axioms that contain multiple test cases. Such axioms can be specified in algebraic specification languages. For example, a metamorphic relation $\forall x, y.(x + y = y + x)$ on integer values of x and y can be written in all algebraic specification languages such as SOFIA (Liu et al., 2014).

Algebraic specifications have been used for test automation since the early 1980s. They have been developed for testing procedural programs (Gonnon et al., 1981; Bernot et al., 1991), object oriented programs (Doong and Frankl, 1994; Hughes and Stotts, 1996; Chen et al., 2001a, 1998a), component-based systems (Kong et al., 2007; Yu et al., 2008), and more recently for service-oriented systems (Liu et al., 2016). The research on metamorphic testing demonstrated that such axioms can be useful for testing software even if they do not form a complete set of axioms, though the latter are often required by test tools that automate software testing from algebraic specifications (Chen et al., 2001a, 1998a).

The main difficulty of applying metamorphic testing is to define the metamorphic relations for the software under test. This is because metamorphic relations are in fact definitions of the semantics of the application. Zhu (2015) proposed a feasible engineering solution via the integration of data mutation testing with metamorphic testing through mutational metamorphic relations (i.e. metamorphisms). This approach is further developed into datamorphic testing (Zhu et al., 2018, 2019a). The test automation environment Morphy shows that the approach can be efficiently implemented and applied. However, datamorphic testing is more general than metamorphic testing. It may contain test morphisms other than metamorphisms. It can also be applied without metamorphic relations as demonstrated by the case study reported in Zhu et al. (2020) and the exploratory strategies studied in this paper, while metamorphic relations is essential for metamorphic testing (Chen et al., 2018).

Research on testing AI applications has been active in recent years (Bai et al., 2018; Gotlieb et al., 2019; Roper and Zhou, 2020). Metamorphic testing is one of the most popular approaches to testing machine learning applications. The testing of driverless vehicles is an interesting application; see for example, Tian et al. (2018) and Zhou and Sun (2019). These works demonstrate that synthetic test cases can find many erroneous behaviours under different realistic driving conditions, many of which led to potentially fatal crashes in three best performing DNNs in the Udacity self-driving car challenge. Most existing testing techniques for DNN-driven vehicles are heavily dependent on the manual collection of test data under different driving conditions. This is prohibitively expensive as the number of test conditions is huge. The works by Tian et al. (2018) and Zhou and Sun (2019) also show that the metamorphic approach can be cost efficient.

Metamorphic testing has also been applied to testing clustering and classification algorithms. Xie et al. (2011) developed a set of metamorphic relations as test oracles for testing such machine learning algorithms. Yang et al. (2019) reported a case study on the use of metamorphic relations to test a clustering function generated by the data mining tool Weka.

It is interesting to observe that datamorphisms are actually used in these cases. For example, DeepTest automatically generates test cases that leverage real-world changes in driving conditions like rain, fog, lighting conditions, etc. via image transformations (Tian et al., 2018). Metamorphic relations are defined based on such image transformations and used to detect erroneous behaviours. Zhu et al. (2019b) reported a case study on the testing of four real industry applications of face recognition. They used feature-editing operators like changing the subject's age, gender, skin tone, make-up etc., to generate synthetic test cases from existing pictures. In Xie et al. (2011) and Yang et al. (2019)'s work, manipulations of datasets were used to test clustering and classification algorithms. The transformations of images and manipulation of datasets are actually datamorphisms.

In general, metamorphic testing differs from the work of this paper because it belongs to confirmatory testing, i.e. it checks that the metamorphic relations are satisfied, rather than discover the behaviour of the software under test. New test cases are usually generated based on existing test cases, where the new test cases are called *follow-up test cases* in the literature. Thus, metamorphic testing replicates the feature of steering in the exploratory testing process. When metamorphic testing is combined with data mutation testing as in the examples discussed above, test designs can be represented in the form of datamorphisms.

6.5. Search-based testing

Search-based testing regards testing as an optimisation problem (Harman et al., 2012; Dave and Agrawal, 2015) to maximise the test effectiveness or test coverage by searching on the space of test cases. Search-based testing techniques can also be applied to ART by considering even spread of test cases as the goal of optimisation.

Genetic algorithms, and other algorithms within evolutionary computing, are often employed to realise such optimisations. In the evolution process, new test cases are generated from existing ones in the population through mutation, crossover and randomisation operators to improve the fitness of the population. Therefore, genetic algorithms provide a steer, just as exploratory test algorithms do. Test design in ET can be represented in the form of the mutation and crossover operators of evolutionary computing, but this fact is rarely studied and used explicitly. Depending on what the fitness metrics encode, a new test case may be executed if it requires information about the program's behaviour. Therefore, search based testing has most of the essential features of ET, but the key difference is in their goals: search-based testing aims to optimise, while ET aims to discover.

6.6. Domain testing

The work reported in this paper is inspired by the domain testing method, in which the input space of the software under test is decomposed into a number of sub-domains according to either the specification or the program code of the software under test. Test cases are then selected on or near to the borders between sub-domains. Domain errors are very common programming errors; for example, sub-domains could be missing and/or the boundaries between sub-domains could be incorrectly implemented. The method of domain testing aims to detect such errors.

The research and practical uses of domain testing can be traced backed to the late 1970s and early 1980s. For example, White and Cohen (1980) studied how programming errors are related to domain modifications and proposed a strategy to select N test cases on the borders and 1 test case near to the borders of the subdomains in order to detect boundary parallel shift errors for linear borders, where N is the dimension of the input space. Similarly, Clarke et al. (1982) proposed a strategy to select N test cases on the border and N test cases nearby. They proved this strategy is capable of detecting both a parallel shift and a rotation of the linear boundary. Afifi et al. (1992) proposed a strategy

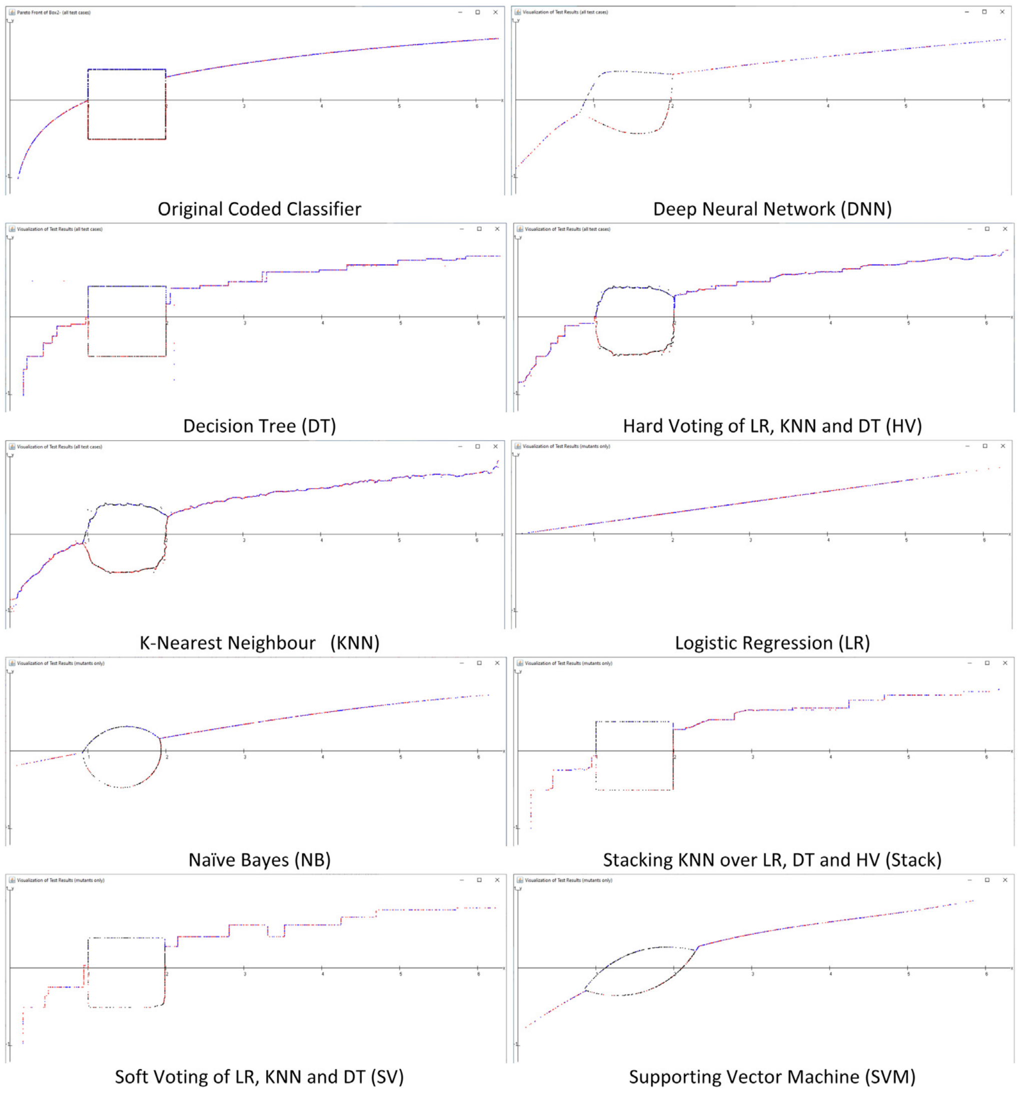


Fig. 19. Subdomain boundaries of various machine learning models of Box 2.

that selects $N + 2$ test cases on and nearby to each border of a subdomain. By applying Zeil's theory of perturbation testing (Zeil, 1983, 1989), they proved that the strategy is capable of detecting linear errors of boundaries defined by non-linear functions (Zeil et al., 1992), where linear errors are linear transformations of the boundary function. A survey of the research on domain testing in the 1980s and 1990s can be found in Zhu et al. (1997). Since then little progress has been reported in the literature.

Domain testing is a typical traditional scripted and confirmatory testing method that derives a complete test set before testing is actually executed and the test results are compared with predetermined expected outputs. There is no immediate execution of test cases after generation, no steering of the testing using the output of previous tests, and the design of test cases is not focused on the variations in the behaviour space. The purpose of domain testing is to confirm that the borders between sub-domains are correctly drawn.

The research on domain testing demonstrated that programming errors often manifest themselves as changes in the boundaries between sub-domains. Test cases on or near those borders are effective at detecting these errors. Errors in a machine learning model must also occur around the borders between sub-domains. It is very useful to know where borders are actually drawn between sub-domains defined by the model. However, the existing domain testing techniques cannot easily be applied to classifiers built from machine learning techniques, because both the expected border as specified and the implemented border as coded are usually not available. Moreover, the domain errors of a machine learning model could be much more complicated than traditional programming errors. Fig. 19 visualises the Pareto fronts of the original coded classifier Box 2 and various machine learning models built from a dataset obtained by a random sampling of Box 2 on 5000 points. It shows that the boundary errors of these machine learning models are highly complicated and significantly different from the coding errors assumed in the research on domain testing.

6.7. Summary of the comparison

To summarise the differences between the proposed approach and the related testing methods discussed above, we contrast these methods on the four essential elements of ET; see Table 8.

7. Conclusion and future work

The Pareto fronts generated by the algorithms studied in this paper contain a huge amount of information about behaviour of the ML models and we are exploring their potential benefits. First, the Pareto front brings a number of possible new ways to analyse and improve ML models. We are currently working on how to use Pareto fronts in the measurement and comparison of ML model's performance.

Another possible benefit is in explaining and/or interpreting the output of a ML model, which has been an active research topic recently; see, for example, Linardatos et al. (2021) and Molnar (2021). Given a Pareto front that represents the borders between classes, a model's classification of a data point could be explained and interpreted, for example, by contrasting it against the nearest points on the surrounding borders and the distance of the point to these boundary points.

The test cases contained in a Pareto front seem also useful to improve model's performance. For example, when the training data is imbalanced, those Pareto front points in minority classes could be used as additional synthetic training data similar to the SMOTE technique (Fernández et al., 2018).

How to present the information contained in Pareto fronts is another interesting topic for future research. For example, the visualisation of Pareto fronts of various ML models in Fig. 19 provides a clear view of their behaviours. An interesting research question for future work is how to visualise models on higher dimensional data spaces. There are a few existing techniques to visualise higher dimension spaces, such as contour charts for visualising 3D models on a 2D space. The effectiveness of such techniques needs to be tested with empirical studies.

There are also many possible variations of the strategies proposed and studied in this paper. In particular, the strategy's algorithms do not need a measurement of the distance between two test cases. However, a distance measurement can be used to decide when to terminate the refinement loop, thereby improving the effectiveness. We are conducting further research on strategies that improve both cost and capability.

This paper focused on multi-class feature-based classifiers whose data spaces are symbolic or numerical values of features and each instance of data is assigned with a single label. A valuable topic for further study is to extend the approach to classifiers on other types of data spaces, such as time series, images, audio and video data, and natural language texts, etc.

Moreover, a machine learning classifier can be:

- *single-labelled*, where one label is assigned to each instance in the data space, thus the classes are non-overlapping sub-domains;
- *multi-labelled*, where multiple labels can be assigned to an instance, thus, a data point may belong to multiple classes and the sub-domains may overlap with each other; and
- *hierarchical*, where labels are organised in a hierarchical structure thus the sub-domain of a class can be divided into a number of subclasses, etc.

In this paper, we have focused on single labelled classifiers. It will be interesting to investigate how to extend the theory and their algorithms to multi-labelled and hierarchical classifiers.

More generally, classifiers are classification models that map from a data space to a set of categorical labels, while predictors are models of functions of continuous or ordered numerical values. Such predictors are often constructed through regression analysis and used for numeric predictions (Aggarwal, 2015). It will also be interesting to adapt the approach studied in this paper to predictors.

Declaration of competing interest

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

Acknowledgements

The work reported in this paper is partially supported by the 2020 Research Excellence Award of Oxford Brookes University, UK. The authors are grateful to the members of the AI Software Engineering research reading group at the School of Engineering, Computing and Mathematics, Oxford Brookes University, for their comments on the drafts of the paper, and discussions at the reading group's activities.

Appendix. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2022.111231.

References

Afifi, F.H., White, L.J., Zeil, S.J., 1992. Testing for linear errors in nonlinear computer programs. In: Proceedings of the 14th International Conference on Software Engineering (ICSE '92). Association for Computing Machinery, New York, NY, USA, pp. 81–91.
Afzal, W., Ghazi, A., Itkonen, J., Torkar, R., Andrews, A., Bhatti, K., 2014. An experiment on the effectiveness and efficiency of exploratory testing. Empir. Softw. Eng. 20, 844–878.
Aggarwal, C., 2015. Data Mining: The Textbook. Springer.
Bach, J., 2002. Exploratory testing. In: van Veenedale, E. (Ed.), The Testing Practitioner. UTN Publishers, Den Bosch, pp. 261–273.
Bach, J., 2003. Exploratory Testing Explained. Technical Report, satisfice.com, Online. URL: http://www.satisfice.com/articles/et-article.pdf.
Bai, X., Li, J., Ulrich, A. (Eds.), 2018. Proc. of IEEE/ACM 13th International Workshop on Automation of Software Test (AST 2018). IEEE Computer Society, Gothenburg, Sweden.
Barr, M., Wells, C., 1989. Category Theory for Computing Science. Prentice Hall.
Bernot, G., Gaudel, M.C., Marre, B., 1991. Software testing based on formal specifications: a theory and a tool. Softw. Eng. J. 387–405.
Bures, M., Frajták, K., Ahmed, B.S., 2018. Tapir: Automation support of exploratory testing using model reconstruction of the system under test. IEEE Trans. Reliab. 67, 557–580.
Chan, K.P., Chen, T.Y., Towey, D., 2005. Adaptive random testing with filtering: An overhead reduction technique. In: Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005). pp. 292–299.
Chen, T.Y., Cheung, S.C., Yiu, S.M., 1998b. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report, Dept. of Computer Science, Hong Kong Univ. of Science and Technology.
Chen, T.Y., Huang, D., Kuo, F.C., 2006. Adaptive random testing by balancing. In: Proceedings of the 1st International Workshop on Random Testing (RT'06). ACM Press, pp. 2–9.
Chen, T.Y., Kuo, F.C., 2007. Is adaptive random testing really better than random testing. In: Proceedings of the 2nd International Workshop on Random Testing 2007 (RT 2007). ACM Press, pp. 64–69.
Chen, T.Y., Kuo, F.C., Liu, H., 2007. On test case distributions of adaptive random testing. In: Proceedings of the 19th Internatio al Conference on Software Engineering and Knowledge Engineering (SEKE'07). pp. 141–144.
Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T.H., Zhou, Z.Q., 2018. Metamorphic testing: A review of challenges and opportunities. ACM Comput. Surv. 51, 4:1–4:27.
Chen, T.Y., Kuo, F.C., Merkel, R.G., Ng, S., 2004a. Mirror adaptive random testing. Inf. Softw. Technol. 46, 1001–1010.
Chen, T.Y., Kuo, F.C., Merkel, R.G., Tse, T.H., 2010. Adaptive random testing: The art of test case diversity. J. Syst. Softw. 83, 60–66.
Chen, T.Y., Leung, H., Mak, I.K., 2004b. Adaptive random testing. In: Proc. of the 9th Asian Computing Science Conference. pp. 320–329.
Chen, H.Y., Tse, T.H., Chen, T.Y., 1998a. In black and white: an integrated approach to class-level testing of object-oriented programs. ACM TOSEM 7, 250–295.
Chen, H.Y., Tse, T.H., Chen, T.Y., 2001a. Taccle: a methodology for object-oriented software testing at the class and cluster levels. ACM Trans. Softw. Eng. Methodol. 10, 56–109.
Chen, T.Y., Tse, T.H., Yu, Y.T., 2001b. Proportional sampling strategy: a compendium and some insights. J. Syst. Softw. 58, 65–81.
Clarke, L., Hassell, J., Richardson, D., 1982. A close look at domain testing. IEEE Trans. Softw. Eng. SE-8, 380–390.
Copeland, L., 2004. A Practitioner's Guide to Software Test Design. Artech House Publishers, Boston and London.

Cortez, P., Cerdeira, A., Almeida, F.T.M., Reis, J., 2009. Modeling wine preferences by data mining from physicochemical properties. Decis. Support Syst. 47, 547–553.
Dave, M., Agrawal, R., 2015. Search based techniques and mutation analysis in automatic test case generation: A survey. In: 2015 IEEE International Advance Computing Conference (IACC). pp. 795–799.
Doong, K., Frankl, P., 1994. The astoot approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. 3, 101–130.
Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Trans. Softw. Eng. SE-10, 438–444.
Eidenbenz, R., Franke, C., Sivanthi, T., Schoenborn, S., 2016. Boosting exploratory testing of industrial automation systems with ai. In: 14th IEEE Conference on Software Testing, Verification and Validation (ICST 2021). pp. 362–371.
Fernández, A., García, S., Herrera, F., Chawla, N.V., 2018. Smote for learning from imbalanced data: Progress and challenges. J. Artificial Intelligence Res. 61, 863–905.
Gebizli, C.S., Sözer, H., 2016. Automated refinement of models for model-based testing using exploratory testing. Softw. Qual. J. 25, 979–1005.
Gebizli, C.Ş., Sozer, H., 2017. Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2017). pp. 23–28. http://dx.doi.org/10.1109/ICSTW.2017.8.
Gonnon, J., McMullin, P., Hamlet, R., 1981. Data-abstraction implementation, specification and testing. ACM Trans. Program. Lang. Syst. 3, 211–223.
Gotlieb, A., Roper, M., Zhang, P. (Eds.), 2019. Proc. of the First IEEE International Conference on Artificial Intelligence Testing (AITest 2019). IEEE Computer Society, Los Alamitos, CA, USA.
Graham, D., van Veenedaal, E., Evans, I., Black, R., 2007. Foundations of Software Testing – ISTQB Certification. Thomason, London.
Hamlet, R., 2002. Random testing. In: Marciniak, J. (Ed.), Encyclopedia of Software Engineering. Wiley, pp. 970–978.
Hamlet, D., Taylor, R., 1990. Partition testing does not inspire confidence. IEEE Trans. Softw. Eng. 16, 1402–1411. http://dx.doi.org/10.1109/32.62448.
Harman, M., Mansouri, A., Zhang, Y., 2012. Search based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45, 11, 61 pages.
Hendrickson, E., 2013. Explore it!. In: The Pragmatic Bookshelf.
Huang, R., Cui, C., Sun, W., Towey, D., 2020. Poster: Is euclidean distance the best distance measurement for adaptive random testing? In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 406–409.
Huang, R., Sun, W., Xu, Y., Chen, H., Towey, D., Xia, X., 2019. A survey on adaptive random testing. IEEE Trans. Softw. Eng..
Hughes, M., Stotts, D., 1996. Daistish: systematic algebraic testing for oo programs in the presence of side-effects. In: Proceedings of ISSTA'96. IEEE CS Press, pp. 53–61.
Itkonen, J., Mäntylä, M.V., 2014. Are test cases needed? replicated comparison between exploratory and test-case-based software testing. Empir. Softw. Eng. 19, 303–342.
Itkonen, J., Mantyla, M.V., Lassenius, C., 2007. Defect detection efficiency: Test case based vs. exploratory testing. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). pp. 61–70.
Itkonen, J., Mantyla, M.V., Lassenius, C., 2013. The role of the tester's knowledge in exploratory software testing. IEEE Trans. Softw. Eng. 39, 707–724.
Itkonen, J., Mäntylä, .M.V., Lassenius, C., 2016. Test better by exploring: Harnessing human skills and knowledge. IEEE Softw. 33, 90–96.
Itkonen, J., Rautiainen, K., 2005. Exploratory testing: a multiple case study. In: Proceedings of International Symposium on Empirical Software Engineering (ISESE 2005). Association for Computing Machinery, pp. 84–93.
Kaner, C., 1988. Testing Computer Software. John Wiley and Sons.
Kaner, C., Falk, J., Nguyen, H.Q., 1999. Testing Computer Software, second ed. John Wiley and Sons.
Kong, L., Zhu, H., Zhou, B., 2007. Automated testing EJB components based on algebraic specifications. In: Proc. of COMPSAC 2007, Vol. 2. pp. 717–722.
Kung, D., Zhu, H., 2009. Software verification and validation.
Linardatos, Papastefanopoulos, V., Kotsiantis, S., 2021. Explainable AI: A review of machine learning interpretability methods. Entropy 23 (18), 1–45.
Liu, D., Wu, X., Zhang, X., Zhu, H., Bayley, I., 2016. Monic testing of web services based on algebraic specifications. In: Proc. of the 10th IEEE International Conference on Service Oriented System Engineering (SOSE 2016). IEEE Computer Society, Oxford, England, UK, pp. 24–33.
Liu, Y., Zhu, H., 2008. An experimental evaluation of the reliability of adaptive random testing methods. In: 2008 Second International Conference on Secure System Integration and Reliability Improvement. pp. 24–31.
Liu, D., Zhu, H., Bayley, I., 2014. Sofia: An algebraic specification language for developing services. In: Proc. of the 8th IEEE International Symposium on Service-Oriented Systems Engineering (SOSE 2014). IEEE Computer Society. IEEE Computer Society Press, Oxford, UK, pp. 70–75.
Loveland, S., Miller, Jr., G.R.P., Shannon, M., 2005. Software Testing Techniques: Finding the Defects that Matter. Charles River Media, Inc., Hingham, Massachusetts, USA.
Makondo, W., Nallanthighal, R., Mapanga, I., Kadebua, P., 2016. Exploratory test oracle using multi-layer perceptron neural network. In: 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI 2016). pp. 1166–1171.
Mao, C., Zhan, X., Chen, J., Chen, J., Huang, R., 2020. Adaptive random testing based on flexible partitioning. IET Softw. 14, 493–505, (12).
Martensson, T., Stahl, D., Martini, A., Bosch, J., 2021. Efficient and effective exploratory testing of large-scale software systems. J. Syst. Softw. 174, 110890.
Mayer, J., 2005. Lattice-based adaptive random testing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05). Association for Computing Machinery, pp. 333–336.
Micallef, M., Porter, C., Borg, A., 2016. Do exploratory testers need formal training? an investigation using hci techniques. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2016). pp. 305–314.
Mohri, M., Rostamizadeh, A., Talwalkar, A., 2012. Foundations of Machine Learning. The MIT Press.
Molnar, C., 2021. Interpretable Machine Learning: A Guide for Making Black Box Models Explainable. URL: https://christophm.github.io/interpretable-ml-book/.
Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing, third ed. Wiley.
Ntafos, S., 1998. On random and partition testing. In: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98). Association for Computing Machinery, New York, NY, USA, pp. 42–48.
Pfahl, D., Yin, H., Mäntylä, M.V., Münch, J., 2014. How is exploratory testing used? a state-of-the-practice survey. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. Association for Computing Machinery, New York, NY, USA, pp. 1–10.
Roper, M., Zhou, Z.Q. (Eds.), 2020. Proc. of the Second IEEE International Conference on Artificial Intelligence Testing (AITest 2020). IEEE Computer Society, Los Alamitos, CA, USA.
Segura, S., Towey, D., Zhou, Z.Q., Chen, T.Y., 2018. Metamorphic testing: testing the untestable. IEEE Softw. (1), http://dx.doi.org/10.1109/MS.2018.2875968.
Shalev-Shwartz, S., Ben-David, S., 2014. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press.
Shan, L., Zhu, H., 2006. Testing software modelling tools using data mutation. In: Proc. of the First IEEE/ACM Workshop on Automation of Software Test (AST 2006). ACM Press, Shanghai, China, pp. 43–49.
Shan, L., Zhu, H., 2009. Generating structurally complex test cases by data mutation: A case study of testing an automated modelling tool. Comput. J. 52, 571–588.
Shoaib, L., Nadeem, A., Akbar, A., 2009. An empirical evaluation of the influence of human personality on exploratory software testing. In: 2009 IEEE 13th International Multitopic Conference. pp. 1–6.
Society, N.A., 1981. Field Guide to North American Mushrooms. Knopf.
Sutton, M., Greene, A., Amini, P., 2007. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley.
Tian, Y., Pei, K., Jana, S., Ray, B., 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE 2018). IEEE Computer Society, Gothenburg, Sweden, pp. 303–314.
Tsoukalas, M., Duran, J., Ntafos, S., 1993. On some reliability estimation problems in random and partition testing. IEEE Trans. Softw. Eng. 19, 687–697.
White, L., Cohen, E.I., 1980. A domain strategy for computer program testing. IEEE Trans. Softw. Eng. SE-6, 247–257.
Whittaker, J.A., 2009. Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design. Pearson Education.
Whittaker, J., Thomason, M., 1994. A markov chain model for statistical software testing. IEEE Trans. Softw. Eng. 20, 812–824.
Xie, X., Ho, J.W.K., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y., 2011. Testing and validating machine learning classifiers by metamorphic testing. J. Syst. Softw. 84, 544–558.
Yang, S., Towey, D., Zhou, Z., 2019. Metamorphic exploration of an unsupervised clustering program. In: Proc. of IEEE/ACM 4th International Workshop on Metamorphic Testing (MET 2019). IEEE Computer Society, pp. 48–54.
Yu, B., Kong, L., Zhang, Y., Zhu, H., 2008. Testing java components based on algebraic specifications. In: Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008). Lillehammer, Norway. pp. 190–199.
Zeil, S., 1983. Testing for perturbations of program statements. IEEE Trans. Softw. Eng. SE-9, 335–346.
Zeil, S., 1989. Perturbation techniques for detecting domain errors. IEEE Trans. Softw. Eng. 15, 737–746.

Zeil, S.J., Afifi, F.H., White, L.J., 1992. Detection of linear errors via domain testing. ACM Trans. Softw. Eng. Methodol. 1, 422–451.

Zhou, Z.Q., Sun, L., 2019. Metamorphic testing of driverless cars. Commun. ACM 62, 61–67.

Zhu, H., 2015. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In: Proc. of the 2nd Int'l Conf. on Trustworthy Systems and their Applications (TSA 2015). IEEE Computer Society, pp. 8–15.

Zhu, H., Bayley, I., 2020. Exploratory datamorphic testing of classification applications. In: Proc. of The 1st IEEE/ACM International Conference on Automation of Software Test (AST 2020). pp. 51–60.

Zhu, H., Bayley, I., Liu, D., Zheng, X., 2019a. Morphy: A Datamorphic Software Test Automation Tool. Technical Report OBU-ECM-AFM-2019-01, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford, UK, URL: http://arxiv.org/abs/1912.09881.

Zhu, H., Bayley, I., Liu, D., Zheng, X., 2020. Automation of datamorphic testing. In: Proc. of 2nd IEEE International Conference on Artificial Intelligence Testing (AITest 2020), pp. 64–72.

Zhu, H., Hall, P., May, J., 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29, 366–427.

Zhu, H., Liu, D., Bayley, I., Harrison, R., Cuzzolin, F., 2018. Datamorphic Testing: A Methodology for Testing AI Applications. Technical Report OBU-ECM-AFM-2018-02, School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK, URL: http://arxiv.org/abs/1912.04900.

Zhu, H., Liu, D., Bayley, I., Harrison, R., Cuzzolin, F., 2019b. Datamorphic testing: A method for testing intelligent applications. In: Proc. of the First IEEE International Conference on Artificial Intelligence Testing (AITest 2019). IEEE Computer Society, Los Alamitos, CA, USA, pp. 149–156.