The Datamorphic Testing Methodology
-- Principles, Tools and Applications to ML

Prof. Hong Zhu
School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk
Acknowledgement and References


Outline

1. Background
2. Principles and Basic Concepts
4. Examples of Application to Machine Learning
Background

- Machine learning (ML) is increasingly used in computer applications
  - Personalisation (e.g. targeted advertisement)
  - Security (e.g. authentication through face recognition, fingerprint, etc.)
  - Driverless vehicles
  - Big Data, IoT, Edge and Fog computing, Cloud computing (e.g. IT Operations)
  - Smart cities, smart homes, healthcare, etc.
  - Robotics (e.g. chatbots, rescue devices, etc.)

- Inadequately tested AI applications impose a threat to the safety, security and reliability of computer systems
  - Fatal accidents of driverless cars
  - Unfairness in recruitment and job applications
  - Etc.

- Testing ML applications are expensive and difficult
  - Large volume of test dataset is required
  - Difficult and expensive to label data for testing
  - Traditional testing techniques, methods and tools are not simply applicable
The Challenges

- **Fundamental differences between traditional programs and ML models**
  - A machine learning model cannot be *debugged*.
    - We cannot change a ML model at microscale manually to fix “bugs”.
    - To improve a ML model, it has to be re-trained!
  - A machine learning model cannot be *verified* or *validated* for its correctness.
    - *Impossible*: In lack of complete verifiable and testable specification of requirements
    - *Undesirable*: Verification or validation of a ML model’s correctness is *undesirable*, if not impossible.
  - *Need to be statistically Assessed*: The quality of a ML model must be *assessed statistically*, because the PAC ML is regarded as the theoretical foundation for ML applications

- **Implications on software testing in practice**
  - How to provide *feedbacks* to developers to improve the quality of the ML model
    - A list of incorrect instances alone (traditional bug reports) may not be useful.
  - How to *assess* the quality of ML models
    - Static testing, such as formal review and Fagan inspection, may not be applicable
    - Quality attributes specific for ML applications: robustness, fairness, etc.
  - How to *manage* testing process and resources
    - Large volume of data and frequent change of the ML model require test automation

17 July 2023  Tutorial on Datamorphic Testing 5
Part 1

Principles and Basic Concepts
The Philosophy of Datamorphic Testing

Datamorphic testing takes a systems engineering approach to software testing.

- It regards software testing as an engineering process.
- It emphasises on the system that embodies testing activities and assets.

Software testing is an engineering process in which a test system is developed, maintained, evolved and operated to achieve the purposes of testing and to manage testing resources effectively and efficiently.

What is a system?
- consisting of components that interact with each other
- demonstrating functions, properties and behaviours that beyond what each individual component alone can
What is a test system

• A test system is a system for supporting testing activities and manage testing resources

Why do we explicitly define a test system

• Specified formally or informally
• Operated to achieve testing purposes
• Tested and formally proved for correctness
• Maintained, reused, and evolved like all software assets
• Implemented as software assets
• Used to achieve test automation

How should a test system be defined and structured

• A test system should not be just an aggregate of unrelated assets.
• A test system should be well structured to enable test automation, especially
  o Effective and efficient performance of testing activities, manage test recourse
  o Easy to evolve when the system under test evolves
  o Reusable for different testing purposes and different systems to be tested
Artefacts of Software Testing

➢ **Entities:**
  - Objects and data used and/or generated in testing
  - *Examples:*
    - *test cases, test suites, the program under test, test design documents, test reports,* etc.

➢ **Morphisms:**
  - Mappings from and/or to test entities
  - Generating and transforming test entities to achieve testing objectives
  - Invoked to perform test activities
  - Composed to form test processes
  - Implemented as test code or test scripts for test automation
  - *Examples:*
    - test case generators, test oracles, test adequacy metrics, test result analysers, bug report generators, etc.
Test Systems in Datamorphic Testing Methodology

**Definition:**
A test system \( T = \langle E, M \rangle \) consists of a set \( E \) of **test entities** and a set \( M \) of **test morphisms**, where each test morphism in \( M \) is a mapping defined on the test entities in \( E \).
Examples of Test Morphisms

- **Seed makers:**
  - Generates a set of test data from other types of test entities. Such test cases are called *seed test cases*.
  - Examples:
    - Generate from the program under test
    - Selected from an existing profile of recorded real data
    - Convert csv files into image, etc.

- **Datamorphisms:**
  - Transforms existing test data to new test data. Such test cases are called *mutants* of the original test data
  - Also called *test data mutation operators* in [Shan & Zhu 2006, Zhu 2015]

- **Metamorphisms:**
  - Predicates on test cases to check if the program is correct or not on the test case
  - Checking the relationship between the original and mutant test data, and their expected outputs from the program
  - **Metamorphic relations** (*compare*):
    - A *k*-ary relation (*k>*1) on test cases
    - A special form of axioms in algebraic formal specifications
Example: Identification of Flowers

- **Datamorphism**: Change background colour to black-and-white

*Seed test case*: original test case

*Mutant test case*: test case generated by applying the datamorphism
Example: More Datamorphisms
Example: A Metamorphism

By changing the background into black-and-white, the flower should be identified as the same kind as in the original picture.

Metamorphic relation:

\[ \text{FlowerRec}(x) = \text{FlowerRec}(\text{ChangeBackground}(x)) \]
4.1 Main Window

Figure 1 below shows Morphy's main user interface. At the top of Morphy's main window are four panels of buttons. This first set of buttons in the Management panel are functions to manage the artefacts of software testing, which include:

a) load a Morphy test specification,
b) load a previously saved test set from a file, which contains intermediate results of testing,
c) save the current test set into a file, which contains the current state of the testing,
d) clean up the system by removing all test cases, messages in the message areas, and the test scripts, etc.

It also gives the class name of the current loaded test specification.

The Activity panel enables the user to perform basic testing activities by invoking various types of test morphisms. These testing activities include the following; see Section 5 for details.

a) Seed: to generate seed test cases using selected seed maker methods;
b) Mutate: to generate mutant test cases using selected datamorphisms;
c) Filter: to remove test cases from the current test set using selected test set filters;
d) Edit test: to show the test cases in the current test pool and to enable manual editing of the test results;
e) Measure: to measure the current test set by invoking the selected test set metrics;

Figure 2. Morphy's Main GUI

(a) Original Photo
(b) With Glasses
(c) Wearing Makeup
(d) Changed Hair Style
(e) b + d
(f) c + d
(g) b + c
(h) b + c + d

Figure 3. Mutants for Face Recognition

In other words, a test set is first order mutant complete if it contains every seed and every first order mutant. A test strategy is to test the software with all the seeds and all the first order mutant test cases generated from the seeds using selected datamorphisms.

The following algorithm generates the minimal test set that is first order mutant complete with respect to a given set of seed test cases and a set of datamorphisms.

Algorithm 1: (Generate 1st Order Mutant Complete Tests)

Input: S = the set of seed test cases;
D = the set of datamorphisms;
Output: C = a set of test cases;

Variables: tempT = temporal set of test cases;

Begin
C = EmptySet;
for (each datamorphism d in D){
tempT = EmptySet;
Assume that d is a k-ary datamorphism;
forall k-tuples (x1,... ,xk) of S {
add d(x1,... ,xk) to tempT;
};
C = C + tempT;
};
return C + S;
End

The following theorem asserts the correctness of the algorithm. The proof can be found in [31].

Theorem 1: The test set generated from S using D by Algorithm 1 is the minimal set of test cases that is first order mutant complete with respect to S and D.

B. Higher Order Mutant Coverage

Datamorphisms can also be applied to test cases multiple times to generate mutants of mutants, which are called higher order mutants and formally defined as follows. For the sake of convenience, a test case $x \in S$ is called a 0'th order mutant of S.

Definition 3: (Higher order mutants)

A test case $y$ is a second order mutant of $S$ by $D$, if there is a $k$-ary datamorphism $d_2 \in D$ and $k$ test cases $x_1, \cdots, x_k$ that:

Example: Datamorphisms

Higher order mutants can be obtained by applying datamorphisms multiple times. They represent combinations of operation conditions/scenarios.
A Category of Test Morphisms

Test Morphisms

Test Generators
- Seed Makers
- Datamorphisms
  - Test case filters
  - Test set filters

Test Oracles
- Relations
  - Unary relations
  - Pre/post-conditions
- Metemorphisms
  - Metamorphic relations

Test Metrics
- Test case metrics
- Test set metrics

Test Executers

Test Result Analysers
- Statistical analyser
  - Visualisation
  - Bug location
Part 2

Morphy: An Automated Tool for Datamorphic Testing
Main Functions

- Management of test systems
  - Management of test entities
  - Management of test specifications

- Test Automation at 3 levels:
  - Activity level:
    - Perform testing activities automatically through invocations of test morphisms
  - Strategy level:
    - Apply test strategies implemented by the tools with user’s selection of parameters
  - Process level:
    - Interactive uses of the tool can be recorded, and replayed
    - Test scripts can be edited and executed

A test system is defined/implemented as a Java class, where
- Test entities are stored in attributes
- Test morphisms are implemented as methods
Graphic User Interface
Test Morphism Panel

- Users can select the test morphisms to apply interactively or as parameters of strategies.

### Seed Makers:
- Name
  - RandomValue10
  - RandomValue100
  - RandomValue200
  - RandomValue400

### Analysers:
- Name
  - saveMessageHead
  - saveMessage
  - visualiseAll
  - visualiseMutants

### Datamorphisms:
<table>
<thead>
<tr>
<th>Name</th>
<th>Arity</th>
<th>Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>rightward</td>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>upward</td>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>mid</td>
<td>2</td>
<td>TooClose</td>
</tr>
<tr>
<td>downward</td>
<td>1</td>
<td>none</td>
</tr>
</tbody>
</table>

### Metamorphisms:
<table>
<thead>
<tr>
<th>Name</th>
<th>Applicable To</th>
<th>Datamorphism</th>
<th>Message</th>
</tr>
</thead>
</table>

### Test Case Metrics:
- Name
  - Distance

### Test Case Filters:
- Name
  - TooClose

### Test Set Metrics:
- Name
  - AvgDistance
  - StdDistance

### Test Set Filters:
- Name
  - sparse

Various types of test morphisms are listed in this panel.

Users can select the test morphisms to apply interactively or as parameters of strategies.
Message Panel

The Message panel shows the activities performed during the test process.
Error Report Panel

The Error Report panel shows errors detected during the test process

Test Error Report:

-- Set zero to Y rule on test case:
{
    id:ac9ffa74-da27-42fb-83ac-3f20cb86dbfd,
    input:<5|0|5>,
    output:isoscelene,
    feature:mutant,
    type:zeroY,
    origins:[ 9e5dc594-1561-4431-8d6a-c086e49c4081],
    correctness:zeroYRule=fail;
}

-- Set zero to Y rule on test case:
{
    id:05bd6e52-1f9c-4b50-b25b-4d51d29e0eed,
    input:<5|0|7>,
}
Management of Test System

- **Load Spec**: Load a test specification that is a Java class file.
- **Load Test Set**: Load a test set file previously save on the computer and add them to the current set of test cases
- **Save Test Set**: Save the current test set to a file
- **Clean**: Re-initialise the system’s state
- **Test Spec Name**: Give the test specification name that is currently used

- When the system is started, it will restore the state of the last time it is used.
- When a new test specification is loaded, the system will initialise its state, so remove all the test cases in the current test set.
Automation at Activity Level

- **Seed**: Invoke selected seed maker morphism(s) to generate a set of seed test cases
- **Mutate**: Apply selected datamorphisms to the current test set to generate mutant test cases and add to the current test set
- **Edit Test**: View and edit the current test set
- **Filter**: Apply selected test set filter(s) to modify the current test set
- **Measure**: Apply selected test set metrics to measure the test quality
- **Execute**: Run the selected executer to run the program under test on the current set of test cases
- **Check**: Check the correctness of the test results against the selected metamorphisms
- **Analyse**: Invoke the selected analysers to generate test report
## View and Edit Test Set

![Test Set Table](image)

<table>
<thead>
<tr>
<th>UUID</th>
<th>Input</th>
<th>Output</th>
<th>Feature</th>
<th>Type</th>
<th>Originals</th>
<th>Correctness</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0322169-be13-4800-922a-20bdc8d929ef</td>
<td>&lt;5.0534850856684750.94412528580332&gt;</td>
<td>blue</td>
<td>original</td>
<td>Type</td>
<td>Originals</td>
<td>Correctness</td>
<td>Distance</td>
</tr>
<tr>
<td>11c4cad-5a2c-4a51-9102-8f3f6b0b0b0c</td>
<td>&lt;5.0534749412403030.94319026974864&gt;</td>
<td>blue</td>
<td>original</td>
<td>Type</td>
<td>Originals</td>
<td>Correctness</td>
<td>Distance</td>
</tr>
<tr>
<td>754c8a8a-b817-4c74-9054-7851b21358da0</td>
<td>&lt;5.09795709810396.83899904738515&gt;</td>
<td>blue</td>
<td>original</td>
<td>Type</td>
<td>Originals</td>
<td>Correctness</td>
<td>Distance</td>
</tr>
<tr>
<td>032a9a9c-9e8c-49d5-bbe1-6c76147e1b</td>
<td>&lt;5.60853632692317.liquor&gt;</td>
<td>blue</td>
<td>original</td>
<td>Type</td>
<td>Originals</td>
<td>Correctness</td>
<td>Distance</td>
</tr>
</tbody>
</table>

[Filter] [Delete] [Save]

---

17 July 2023  Tutorial on Datamorphic Testing  25
Three sets of test strategies have been implemented:
- Mutant combination strategies
- Exploratory strategies
- Test optimisation strategies using genetic algorithms

The user selects a strategy from the drop down menu, select the parameters as instructed, then press the execute button to run the selected test strategy.

The execution process will be reported in the message panel.
Automation at Process Level

Create a new test script
Load a previously saved test script
Start recording interactive test activities
Play the current test script
Save the test script to a file
View test script
Clean the test script
Example of Test Script

Test Script Name: FirstScript

```plaintext
clean()
makeSeed([RandomValue1000])
makeSeed([RandomValue1000])
randomWalk([downward, upward, rightward, leftward, mid];10;1000)
 analyse([visualiseAll, visualiseMutants, statistics])
```
The test scripting facility enables interactive test specification to be invoked. It also implements various test morphisms as implemented in Morphy.

<table>
<thead>
<tr>
<th>Test Tool Morphy</th>
<th>Section 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>The architecture of Morphy consists of three main facilities:</td>
<td></td>
</tr>
<tr>
<td>Analyser</td>
<td>Test Executer</td>
</tr>
<tr>
<td>Test Set Filter</td>
<td>Test Set Metrics</td>
</tr>
<tr>
<td>Test Case Filter</td>
<td>Test Case Metrics</td>
</tr>
<tr>
<td>Metamorphism</td>
<td>Datamorphism</td>
</tr>
<tr>
<td>Seed Maker</td>
<td>Morphism</td>
</tr>
</tbody>
</table>

The test set management facility enables test sets to be saved into files, loaded from files and edited in a graphic user interface. The test runner enables test specifications to be run, and from the test results, the test set metrics can be evaluated. The test case filter enables test cases to be selected from the test set, and the test case metrics can be evaluated.

Table 1. Annotations of Test Morphisms

<table>
<thead>
<tr>
<th>Test Morpism</th>
<th>Annotation</th>
<th>Repository</th>
<th>Test Spec</th>
<th>Test Set Management</th>
<th>Test Set Repository</th>
<th>Test Set Editor</th>
<th>Test Case Filter</th>
<th>Test Case Metrics</th>
<th>Test Set Filter</th>
<th>Test Set Metrics</th>
<th>Test Runner</th>
<th>Test Spec (Bytecode)</th>
<th>Java IDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;str&quot;</td>
<td>Nil</td>
<td>Input</td>
<td>Nil</td>
<td>Nil</td>
<td>Real</td>
<td>Boolean</td>
<td>TestCase</td>
<td>Boolean</td>
<td>TestCase</td>
<td>Boolean</td>
<td>Void</td>
<td>Return</td>
<td>Void</td>
</tr>
</tbody>
</table>

The uses of various types of test entities and morphisms are listed as follows.

- **Analyser**: Act as an analyser, where the testing artefacts can be managed, basic testing activities can be performed, and automated testing facilities can be invoked. A wizard has been developed as an Eclipse plugin to but a wizard has been developed as an Eclipse plugin to.

- **Test Executer**: Use the TestExecuter method to run testing on the Under Test Program. The Under Test Program is a Java class, which de-...
Morphy’s Format to Specify Test Systems

- **Test Entities:**
  - Java generic class `TestCase` for representing test cases
  - Java generic class `TestPool` for representing test suites/set

- **Test Morphisms:**
  - Java methods annotated with metadata

<table>
<thead>
<tr>
<th>Morphism</th>
<th>Annotation</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed Maker</td>
<td>@SeedMaker</td>
<td>Nil</td>
<td>Void</td>
</tr>
<tr>
<td>Datamorphism</td>
<td>@Datamorphism</td>
<td>TestCase</td>
<td>TestCase</td>
</tr>
<tr>
<td>Metamorphism</td>
<td>@Metamorphism</td>
<td>TestCase</td>
<td>Boolean</td>
</tr>
<tr>
<td>Test Case Metrics</td>
<td>@TestCaseMetrics</td>
<td>TestCase</td>
<td>Real</td>
</tr>
<tr>
<td>Test Case Filter</td>
<td>@TestCaseFilter</td>
<td>TestCase</td>
<td>Boolean</td>
</tr>
<tr>
<td>Test Set Metrics</td>
<td>@TestSetMetrics</td>
<td>Nil</td>
<td>Real</td>
</tr>
<tr>
<td>Test Set Filter</td>
<td>@TestSetFilter</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>Test Executer</td>
<td>@TestExecuter</td>
<td>Input</td>
<td>Output</td>
</tr>
<tr>
<td>Analyser</td>
<td>@Analyser</td>
<td>Nil</td>
<td>Void</td>
</tr>
</tbody>
</table>
Example 1. Triangle Classification

- **The Program under Test:**
  
  "reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral."


- **The Problem of Testing:**
  Myer listed 14 questions for testers to assess how well he/she tests the program for such a seemly simple program and reported that "highly qualified professional programmers score, on the average, only 7.8 out of a possible 14".

- **The Research Questions:**
  - Can datamorphic testing achieve a good score?
  - Can Morphy automate this testing process?
Structure of the Test System

@TestSetContainer(
    inputTypeName = "Triangle",
    outputTypeName = "TriangleType"
)
public TestPool<Triangle, TriangleType> testSuite = new TestPool<Triangle, TriangleType>();
package morphy.examples;
public class Triangle {
    public int x =0;
    public int y =0;
    public int z =0;

    public Triangle() {
        x=0; y=0; z=0;
    }

    public Triangle(int a, int b, int c) {
        x=a; y=b; z=c;
    }

    public String toString() {
        String str = "<"+x+"|"+y+"|"+z+">";
        return str;
    }

    public void valueOf(String str) { ... }
}
Seed Makers

- Four methods were coded to generate seed test cases
  - Literal constants without expected output
  - Literal constants with expected output
  - Manual input
  - Read test cases from a file

Example:

```java
@MakeSeed
public void makeSeeds(){
    testSuite.addInput(new Triangle(5,5,5));
    testSuite.addInput(new Triangle(5,5,7));
    testSuite.addInput(new Triangle(5,7,9));
    testSuite.addInput(new Triangle(3,5,9));
}
```
Datamorphisms

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>increaseX</td>
<td>Increase the value of x by 1</td>
</tr>
<tr>
<td>increaseY</td>
<td>Increase the value of y by 1</td>
</tr>
<tr>
<td>increaseZ</td>
<td>Increase the value of z by 1</td>
</tr>
<tr>
<td>decreaseX</td>
<td>Decrease the value of x by 1</td>
</tr>
<tr>
<td>decreaseY</td>
<td>Decrease the value of y by 1</td>
</tr>
<tr>
<td>decreaseZ</td>
<td>Decrease the value of z by 1</td>
</tr>
<tr>
<td>swapXY</td>
<td>Swap the values of x and y</td>
</tr>
<tr>
<td>swapXZ</td>
<td>Swap the values of x and z</td>
</tr>
<tr>
<td>swapYZ</td>
<td>Swap the values of y and z</td>
</tr>
<tr>
<td>rotateL</td>
<td>Rotate the values of x, y and z left</td>
</tr>
<tr>
<td>rotateR</td>
<td>Rotate the values of x, y and z right</td>
</tr>
<tr>
<td>copyXToY</td>
<td>Copy the value of x to y</td>
</tr>
<tr>
<td>copyXToZ</td>
<td>Copy the value of x to y</td>
</tr>
<tr>
<td>copyYToZ</td>
<td>Copy the value of y to z</td>
</tr>
<tr>
<td>negateX</td>
<td>Negate the value of x</td>
</tr>
<tr>
<td>negateY</td>
<td>Negate the value of y</td>
</tr>
<tr>
<td>negateZ</td>
<td>Negate the value of z</td>
</tr>
<tr>
<td>zeroX</td>
<td>Set the value of x to 0</td>
</tr>
<tr>
<td>zeroY</td>
<td>Set the value of y to 0</td>
</tr>
<tr>
<td>zeroZ</td>
<td>Set the value of z to 0</td>
</tr>
</tbody>
</table>
Example of Datamorphism

```java
@Datamorphism
public TestCase<Triangle, TriangleType> increaseX(TestCase<Triangle, TriangleType> seed){
    TestCase<Triangle, TriangleType> mutant = new TestCase<Triangle, TriangleType>();
    Triangle m = new Triangle(1,1,1);
    m.x=seed.input.x+1;
    m.y=seed.input.y;
    m.z=seed.input.z;
    mutant.input = m;
    return mutant;
}
```
Metamorphisms

- There is a metamorphism for test cases generated by the literal constant with expected output to compare the execution results against the expected output.
- For each datamorphism, there is a corresponding metamorphism to check correctness of the test output on the mutant test cases.

```java
@Metamorphism(
    applicableTestCase="mutant",
    applicableDatamorphism = "increaseX",
    message="Increase on Parameter X rule."
)

public boolean increaseXRule(TestCase<Triangle, TriangleType> x) {
    String originalId = x.getOrigins().get(0);
    TestCase originalTc = testSuite.get(originalId);
    if (originalTc.output == TriangleType.equilaterial){
        return (x.output == TriangleType.isoscelene);
    }
    return true;
}
```
package morphy.examples;

import morphy.annotations.*;

public class TriangleTest1 extends TriangleTestSpec {
    @TestExecutor
    public TriangleType TriangleClassifier1(Triangle tc) {
        int x = tc.x;
        Triangle1 tx = new Triangle1(x, tc.y, tc.z);
        return tx.Classify();
    }
}

package morphy.examples;

class Triangle1 {
    public int x, y, z;
    public Triangle1(int a, int b, int c) {
        x = a; y = b; z = c;
    }
    public TriangleType Classify() { ... }
}

The code under test
Test Result Analyser

An analyser was written to analyse the test results on a test set statistically.

```java
@Analyser
public void statisticsOfCorrectness() { ... }
```

An output of the analyser
Example 2: Trigonometric Functions

- **Programs under Test:**
  - Three trigonometric functions $\sin(x)$, $\cos(x)$ and $\tan(x)$ provided by Java math library

- **Problem of Test:**
  - Can such functions be tested for their accuracy and correctness?

- **Solution in the damomorphic testing approach:**
  - Test on specific input values that the output value is known
  - Test on random input test values to check if algebraic laws of these functional are held
    - Laws involve multiple invocation of the same function
    - Laws invoke multiple functions on the same input values
    - Laws invoke multiple functions on different input values
Two seed makers are written:
1. Generate a number of random real numbers (without expected outputs)
2. Generate a set of special input values and the corresponding expected output

Special Input Values and Expected Outputs

<table>
<thead>
<tr>
<th>$x$</th>
<th>0</th>
<th>$\frac{\pi}{6}$</th>
<th>$\frac{\pi}{4}$</th>
<th>$\frac{\pi}{3}$</th>
<th>$\frac{\pi}{2}$</th>
<th>$\frac{2\pi}{3}$</th>
<th>$\frac{3\pi}{4}$</th>
<th>$\frac{5\pi}{6}$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sin(x)$</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{\sqrt{2}}{2}$</td>
<td>$\frac{\sqrt{3}}{2}$</td>
<td>1</td>
<td>$\frac{\sqrt{3}}{2}$</td>
<td>$\frac{\sqrt{2}}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>0</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>1</td>
<td>$\frac{\sqrt{3}}{2}$</td>
<td>$\frac{\sqrt{2}}{2}$</td>
<td>$\frac{1}{2}$</td>
<td>0</td>
<td>$-\frac{1}{2}$</td>
<td>$-\frac{\sqrt{2}}{2}$</td>
<td>$-\frac{\sqrt{3}}{2}$</td>
<td>-1</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>0</td>
<td>$\frac{\sqrt{3}}{3}$</td>
<td>1</td>
<td>1</td>
<td>$\infty$</td>
<td>$-\sqrt{3}$</td>
<td>-1</td>
<td>$-\frac{\sqrt{3}}{3}$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\frac{\pi}{6}$</th>
<th>$\frac{5\pi}{6}$</th>
<th>$\frac{4\pi}{3}$</th>
<th>$\frac{3\pi}{2}$</th>
<th>$\frac{5\pi}{3}$</th>
<th>$\frac{7\pi}{4}$</th>
<th>$\frac{11\pi}{6}$</th>
<th>$2\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sin(x)$</td>
<td>$-\frac{1}{2}$</td>
<td>$-\frac{\sqrt{2}}{2}$</td>
<td>$-\frac{\sqrt{3}}{2}$</td>
<td>$-1$</td>
<td>$-\frac{\sqrt{3}}{2}$</td>
<td>$-\frac{\sqrt{2}}{2}$</td>
<td>$-\frac{1}{2}$</td>
<td>0</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>$-\frac{\sqrt{3}}{2}$</td>
<td>$-\frac{\sqrt{2}}{2}$</td>
<td>$-\frac{1}{2}$</td>
<td>0</td>
<td>$\frac{1}{2}$</td>
<td>$\frac{\sqrt{2}}{2}$</td>
<td>$\frac{\sqrt{3}}{2}$</td>
<td>1</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>$\frac{\sqrt{3}}{3}$</td>
<td>1</td>
<td>$\sqrt{3}$</td>
<td>$\infty$</td>
<td>$-\sqrt{3}$</td>
<td>-1</td>
<td>$-\frac{\sqrt{3}}{3}$</td>
<td>0</td>
</tr>
</tbody>
</table>
## Datamorphisms

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>halfPiPlus</td>
<td>$x \rightarrow \pi/2 + x$</td>
<td>halfPiMinus</td>
<td>$x \rightarrow \pi/2 - x$</td>
</tr>
<tr>
<td>piPlus</td>
<td>$x \rightarrow \pi + x$</td>
<td>piMinus</td>
<td>$x \rightarrow \pi - x$</td>
</tr>
<tr>
<td>twoPiPlus</td>
<td>$x \rightarrow 2\pi + x$</td>
<td>twoPiMinus</td>
<td>$x \rightarrow 2\pi - x$</td>
</tr>
<tr>
<td>sum</td>
<td>$(x, y) \rightarrow x + y$</td>
<td>diff</td>
<td>$(x, y) \rightarrow x - y$</td>
</tr>
<tr>
<td>negate</td>
<td>$x \rightarrow -x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example: Implementation of datamorphisms

```java
@Datamorphism
def TestCase<Double, Trigonometrics> PiMinus(TestCase<Double, Trigonometrics> seed){
    TestCase<Double, Trigonometrics> mutant = new
    TestCase<Double, Trigonometrics>();
    mutant.input = Math.PI - seed.input;
    return mutant;
}
```

17 July 2023   Tutorial on Datamorphic Testing  42
### Algebraic laws of Trigonometric Functions

\[
\begin{align*}
\sin(x + y) &= \sin(x)\cos(y) + \cos(x)\sin(y) & \cos(x + y) &= \cos(x)\cos(y) - \sin(x)\sin(y) \\
\sin(x - y) &= \sin(x)\cos(y) - \cos(x)\sin(y) & \cos(x - y) &= \cos(x)\cos(y) + \sin(x)\sin(y) \\
\tan(x + y) &= \frac{\tan(x) + \tan(y)}{1 - \tan(x)\tan(y)} & \tan(x - y) &= \frac{\tan(x) - \tan(y)}{1 + \tan(x)\tan(y)} \\
\sin(\pi + x) &= -\sin(x) & \cos(\pi + x) &= -\cos(x) & \tan(\pi + x) &= \tan(x) \\
\sin(\pi - x) &= \sin(x) & \cos(\pi - x) &= -\cos(x) & \tan(\pi - x) &= -\tan(x) \\
\sin(\pi/2 + x) &= \cos(x) & \cos(\pi/2 + x) &= -\sin(x) & \tan(\pi/2 + x) &= -1/\tan(x) \\
\sin(\pi/2 - x) &= \cos(x) & \cos(\pi/2 - x) &= \sin(x) & \tan(\pi/2 - x) &= 1/\tan(x) \\
\sin(2\pi - x) &= -\sin(x) & \cos(2\pi - x) &= \cos(x) & \tan(2\pi - x) &= -\tan(x) \\
\sin(2\pi + x) &= \sin(x) & \cos(2\pi + x) &= \cos(x) & \tan(2\pi + x) &= \tan(x) \\
\sin(-x) &= -\sin(x) & \cos(-x) &= \cos(x) & \tan(-x) &= -\tan(x)
\end{align*}
\]
Examples: Implementations of Metamorphisms

@Metamorphism(
    applicableTestCase="seed",
    message="Special Sin(x) value"
) public boolean specialSinValueAssertion(TestCase<Double, Trigonometrics> tc) {
    if (expected.get(tc.id).output == null) { return true; };
    return (Math.abs(tc.output.sin - expected.get(tc.id).output.sin) < error);
}

@Metamorphism(
    applicableTestCase="mutant",
    applicableDatamorphism="HalfPiPlus",
    message="The rule: Sin(pi/2+x) = Cos(x)"
) public boolean HalfPiPlusSinAssertion(TestCase<Double, Trigonometrics> tc) {
    TestCase<Double, Trigonometrics> originalTc = testSuite.get(tc.getOrigins().get(0));
    return (Math.abs(tc.output.sin - originalTc.output.cos) <= error);
}
Test Result Analysers

- Two test result analysers:
  - Statistical analysis of test result:
    - Reused (simplified) a part of the analyser developed for Triangle Classification case study
  - Visual display of the functions

### Statistics:
- Total number of test cases = 44400
- Number of original test cases = 37
- Number of mutant test cases = 44363
- Number of test cases checked correctness = 44380
- Number of times pass checking = 131866
- Number of times failed checking = 1274
- Failure rate = 0.9568874868559412%
Part 3

Test Strategies for Machine Learning Applications

1. Scenario-Based Confirmatory Testing
2. Exploratory Testing of ML Classifiers
3. Scenario-based Exploratory Functional Testing
Scenario-Based Confirmatory Testing

1) Development process of test systems
2) Strategies to combine scenarios and adequacy criteria
3) Algorithms to generate adequate test sets
Test System Development Process

Stage 1: Analysis
Analysis of the testing problem to design a test system
- Identify the seed test cases
- Identify the datamorphisms
- Identify the metamorphisms

Stage 2: Realisation
Realisation of the elements in the test system
- Collecting seed test data
- Implementation of datamorphisms
- Implementation of metamorphisms

Stage 3: Execution
Execution of testing using the test system
- Selection of test adequacy criteria
- Generating test cases
- Execution of test
Stage 1: Analysis

Analysis Stage

Analysing the operation conditions of the application

Normal/Perfect operation conditions

Defining Seed test cases

Specification of seed test cases

Abnormal operation conditions

Deriving&Defining metamorphisms

Specification of metamorphisms

Defining datamorphisms

Specification of datamorphisms

Definition 4:
(Datamorphic Test Framework)
Let $D$ be the input domain of a program $P$ under test. A datamorphic test framework $F$ is an ordered triple $\langle S, \Psi, M \rangle$, where $S \subseteq D$ is a finite subset of $D$. The elements of $S$ are called the seed test cases, or simply seeds. $\Psi$ is a finite set of datamorphisms, and $M$ is a finite set of metamorphisms. 

III. Testing Process and Strategies

A. Process of Datamorphic Testing

As illustrated in Fig. 2, the datamorphic testing process consists of three stages.

1) Stage 1: Analysis:
The first stage is analysis of the testing problem in order to design a datamorphic test framework. In this stage, seed test cases, datamorphisms and metamorphisms are identified. These three elements are closely related to each other, thus should be engineered systematically. Analysis starts by identifying the operating conditions of the application. For a face recognition application at an international airport's border control, for example, the input to the software is a photo from a camera fitted on an automatic passport checking machine and the photo of the passport holder.
Example: Face Recognition (1)

- Usage 1: Automated passport control at airport
- Usage 2: Detect criminal suspects using images from surveillance cameras

<table>
<thead>
<tr>
<th>Operation Conditions</th>
<th>Usage 1</th>
<th>Usage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front face images in database</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Side face images in database</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image of older age</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image sun tanned</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image in different hair style/colour</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image wearing makeup</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image with sunglasses</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image with beard</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Variable lighting and background</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image from a side angle</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Face image from an upper angle</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Image from artist drawings</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Example: Face Recognition (2)

- **Seed test cases:**
  - A set of photos of human faces in different races, ages, genders, etc.
- **Datamorphisms:**
  - Add a pair of glasses;
  - Add makeup;
  - Change the background;
  - Change the illumination;
  - Change hair style;
  - Change hair colour;
  - Swap: replace a part of the image with another person’s image.
- **Metamorphisms:**

  \[
  \text{FaceSimile}(x, \varphi(x)) \geq 80\%
  \]

  \[
  \text{FaceOf}(\text{Swap}(x, y)) = \text{FaceOf}(x) \quad \text{and} \quad \text{FaceOf}(\text{Swap}(x, y)) = \text{FaceOf}(y)
  \]

  \(\varphi(x)\) is any of the datamorphisms given in the Table
  
  \(\text{FaceSimile}\) is any of the face recognition application under test

  Depends on the application

---

17 July 2023

Tutorial on Datamorphic Testing
Stage 2: Realisation

- Seeds:
  - Often available from other development activities, such as training data, benchmarks
  - Can be collected from the real world, though costly
  - Could be manual effort
- Datamorphisms:
  - Often can be implemented as small program code fairly easily
  - Many application domains have open source, library, etc. available
- Metamorphisms:
  - Often easy to implement as small program code fairly easily
**Example: Datamorphisms of Images**

- **Seed test case (a):**
  A photo in the Public dataset *Labeled Faces in the Wild at URL:*
  http://vis-www.cs.umass.edu/lfw/

- **Mutants: (b – j)**
  A subset of the photos obtained from the seed by manipulations of the seed photo.
Implementation of The Datamorphisms

AttGAN’s Face Attribute Editing Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bald</td>
<td>Change the facial image into bald</td>
</tr>
<tr>
<td>Bangs</td>
<td>Add bangs to the facial image</td>
</tr>
<tr>
<td>Black Hair</td>
<td>Change the hair colour into black</td>
</tr>
<tr>
<td>Blond Hair</td>
<td>Change the hair colour into blond</td>
</tr>
<tr>
<td>Brown Hair</td>
<td>Change the hair colour into brown</td>
</tr>
<tr>
<td>Bushy Eyebrows</td>
<td>Change the eyebrows to be bushy</td>
</tr>
<tr>
<td>Eyeglasses</td>
<td>Add eyeglasses to the image</td>
</tr>
<tr>
<td>Male</td>
<td>Change the image from female to male</td>
</tr>
<tr>
<td>Mouth Open</td>
<td>Change the mouth to be slightly open</td>
</tr>
<tr>
<td>Mustache</td>
<td>Add or remove mustache to the facial image</td>
</tr>
<tr>
<td>Beard</td>
<td>Add or remove beard</td>
</tr>
<tr>
<td>Pale Skin</td>
<td>Make the skin tone to be pale</td>
</tr>
<tr>
<td>Young</td>
<td>Change the image to look younger</td>
</tr>
</tbody>
</table>
Stage 3: Execution

For example, as stated earlier, it is possible to edit an image and generate a mutant of an image as shown in Fig. 1. Then, by feeding both the original and the mutant images to a facial recognition application, the correctness of the application can be checked by comparing the outputs of these two test cases. If the outputs are identical, the application passes the test; otherwise, an error is detected.

C. Seed Test Cases

For a datamorphism to be useful, we must have a set of known test cases, called the seed test cases, or simply seeds. In the face recognition example, a seed could be an image of a person's face. Such a set of seeds is normally available for testers of many AI applications, for example, as training data for an ML application. The seeds could be a subset of such training data selected at random or according to certain criteria. However, seeds alone are inadequate. Our method uses the seeds to generate more test cases to make an adequate test of the application. Seeds are not necessarily labeled with the expected outputs unless the datamorphisms require such labels.

A datamorphism may well be applicable to mutants, especially when the mutants are generated by a different datamorphism. For example, in Fig. 1, (h) is obtained by applying a datamorphism on mutant (e).

In summary, our testing framework consists of three elements: a set of seed test cases, a set of datamorphisms and a set of metamorphisms.

Definition 4: (Datamorphic Test Framework)

Let D be the input domain of a program P under test. A datamorphic test framework F is an ordered triple ⟨S, Ψ, M⟩, where S ⊆ D is a finite subset of D. The elements of S are called the seed test cases, or simply seeds. Ψ is a finite set of datamorphisms, and M is a finite set of metamorphisms.

The next section discusses how to construct a datamorphic test framework and how such a test framework can be used with different strategies.

III. Testing Process and Strategies

A. Process of Datamorphic Testing

As illustrated in Fig. 2, the datamorphic testing process consists of three stages.

1) Stage 1: Analysis:

The first stage is analysis of the testing problem in order to design a datamorphic test framework. In this stage, seed test cases, datamorphisms and metamorphisms are identified. These three elements are closely related to each other, thus should be engineered systematically. Analysis starts by identifying the operating conditions of the application. For a face recognition application at an international airport's border control, for example, the input to the software is a photo from a camera fitted on an automatic passport checking machine and the photo of the passport holder.
Mutant Combination Strategies

**Basic Ideas**
- Uses seed test cases to test the normal operation condition of the AI system under test,
- Uses datamorphisms to transform a test case that represents other operation conditions that can be derived from the normal operation conditions.
- Combining datamorphisms means combinations of different operation conditions.

**Examples**
- For testing face recognition applications: datamorphisms are used to transform the images of human faces by editing the facial attributes, such as adding makeup, wearing glasses, changing skin tunes, change hair styles and colour, etc.
- For testing driverless vehicles in [Tian et al. 2018], datamorphisms are developed to alter the weather condition of a recorded driving process to be in fog, to transform the lighting condition from daytime to night time with street lights, etc.
First Order Mutant Coverage

The Notion of First Order Mutants

- First order mutants are mutant test cases generated from seed test cases.
- Each first order mutant represents one operation condition of the system.

Let $T$ be the set of all possible test cases for the software under test, $S \subseteq T$ ($S \neq \emptyset$) be a set of test cases, and $D \neq \emptyset$ be a set of datamorphisms and $d \in D$ be a datamorphism in $D$. We say that $d$ is $k$-ary ($k > 0$), if $d : T^k \rightarrow T$.

**Definition 1**: (First Order Mutants)

A test case $y \in T$ is called a first order mutant test case, or simply a first order mutant, of $S$ generated by $D$, if there is a $k$-ary datamorphism $d \in D$ and test cases $x_1, \ldots, x_k \in S$ such that $y = d(x_1, \ldots, x_k)$. □
Test Adequacy Criterion: First Order Mutant Completeness

Definition 2: (First Order Mutant Completeness)
A set $C$ of test cases is first order mutant complete with respect to $S$ and $D$, if $S \subseteq C$, and for each $d : T^k \rightarrow T \in D$, and each $x_i \in S$, $i = 1, \ldots, k$, there is a test case $y \in C$ such that $y = d(x_1, x_2, \ldots, x_k)$, where $d$ is $k$-ary.

- A test set is first order mutant complete means it contains all seed test cases and all first order mutants of the seed test cases
- Testing on a test set that is first order mutant complete means the testing covered all operation conditions, but not their combinations.
Algorithm 1: Generate 1st Order Complete Test Set

Input: S = the set of seed test cases; D = the set of datamorphisms;
Output: C = a set of test cases;
Variables: tempT = temporal set of test cases;

Begin

1: C = EmptySet;
2: for (each datamorphism d in D){
2.1: tempT = EmptySet;
2.2: Assume that d is a k-ary datamorphism;
2.3: forall k-tuples (x1,...,xk) of S {
    add d(x1,...,xk) to tempT;
}
2.4: C = C + tempT;
};
3: return C + S;
End
Correctness of The Algorithm

**Theorem 1** The test set generated from $S$ using $D$ by Algorithm 1 is the minimal set of test cases that is first order mutant complete with respect to $S$ and $D$.

*Proof.*

(a) *Completeness*: Assume that the output test set $C$ from Algorithm 1 is not complete. This means there is either a seed test case $y$ not in $C$ or there is a first order mutant $y$ generated from seeds $x_1, \ldots, x_k \in S$ by using a $k$-ary datamorphism $d \in D$ is not in $C$. In the former case, it is in conflict with Step 3. In the latter case, it is in conflict with Step 2.3. Therefore, the assumption is incorrect.

(b) *Minimalness*: It is obvious to see that the output only contains seeds and first order mutants. $\square$
Example:

The application under test:
- classify points in a two-dimensional space into three types: red, blue and black.

The seed test set:
- 100 random points

Datamorphism:
- Generates the midpoint of two test cases.

The test set generated:
- 10000 points as the 1st order mutant test cases
- 100 original test cases
High Order Mutants

Definition 3 (Higher order mutants)

A test case \(y\) is a second order mutant of \(S\) by \(D\), if there is a \(k\)-ary datamorphism \(d \in D\) and \(k\) test cases \(x_1, \ldots, x_k\) such that
\[
y = d(x_1, \ldots, x_k)
\]
and for all \(x_i\), \(x_i\) is either in \(S\) or a first order mutant of \(S\) by \(D\), and at least one of \(x_1, \ldots, x_k\) is a first order mutant of \(S\) by \(D\).

A test case \(y\) is an \(n\)'th order mutant of \(S\) by \(D\) (\(n > 1\)), if there is a \(k\)-ary datamorphism \(d \in D\) and \(k\) test cases \(x_1, \ldots, x_k\) such that
\[
y = d(x_1, \ldots, x_k)
\]
and \(x_i\) are \(m\)'th order mutants of \(S\) by \(D\), where \(m < n\), and at least one of \(x_1, \ldots, x_k\) is a \((n - 1)\)'th order mutant of \(S\) by \(D\). \(\square\)
Examples of 1st Order and Higher Order Mutants

Higher order mutants are obtained by applying datamorphisms multiple times. They represent combinations of operation conditions.
Test Adequacy Criterion: $K$’th Order Mutant Completeness

**Definition 4:** (K’th order mutant completeness) A set $C$ of test cases is $k$’th order mutant complete with respect to $S$ and $D$, if it contains all $i$’th order mutant test cases of $S$ by $D$ for all $i = 0, \ldots, k$. □

**Algorithm:** Generate K’th Order Mutant Complete Test Sets:
- Call Algorithm 1 repeatedly for $K$ times with the previous output as the input of the next call

**Correctness of the Algorithm:**

*Corollary of Theorem 1:* By repeating Algorithm 1 for $K$ times such that each time uses the output test set as the input to the next invocation of the algorithm, the result test set is the minimal $K$’th order mutant complete. □
Permutation Completeness and Exhaustive Test

Assume that the set $D$ of datamorphisms contains $N$ methods.

- **Permutation complete test set:**
  
  If a test set is $N$th order mutant complete with respect to $S$ and $D$, it will contain all permutations of the datamorphisms applied to all test cases.

- **Exhaustive test set:**
  
  If the datamorphisms are *associative*, *commutative*, *distributive* and *idempotent*, a permutation complete test set contains all possible test cases that can be derived from a given set of test cases using the set of datamorphisms. The test set is therefore *exhaustive* with regard to the set of seeds and the datamorphisms.
Combinations of Datamorphisms

- 3 datamorphisms: \(d_1, d_2, d_3\)
- 2 seed test cases: \(s_1, s_2\)

(a) and (b) are 1st order
(c) - (f) are 2nd order
There are more possible combinations
Test Adequacy Criterion: Combinatorial Coverage

**Definition 5**: (Combinatorial Coverage)

A set $\mathcal{C}$ of datamorphism combinations is *combinatorial complete* for $D$, if for all non-empty subsets $D' \subseteq D$, there is a combination $c \in \mathcal{C}$ such that $D'$ is the set of datamorphisms in $c$.

A set $C$ of test cases is *combinatorial complete* with respect to $S$ and $D$, if

- there is a set $\mathcal{C}$ of datamorphism combinations that is combinatorial complete with respect to $D$; and
- for every combination $c \in \mathcal{C}$, if $c$ is $k$-ary, then for all $k$-tuples of test cases $(x_1, \cdots, x_k) \in S^k$, there is a test case $y$ in $C$ such that $y = c(x_1, \cdots, x_k)$. □
Algorithm 2: Generate Combinatorial Complete Test Set

Input: $S =$ the set of seed test cases;
       $D =$ the set of datamorphisms;
Output: $C =$ a set of test cases;
Variables: tempT = temporal set of test cases;

Begin
1:    for (each datamorphism $d$ in $D$) {
1.1:    tempT = empty_set;
1.2:    Assume $d$ is a $k$-ary, where $k > 0$;
1.3:    for (all $k$-tuples $(x_1,\ldots,x_k)$ of $S$) {
                add $d(x_1,\ldots,x_k)$ to tempT;
           }
1.4:    $S = S + \text{tempT}$;
    }
2:    return $C + S$;
End
Correctness of the Algorithm

**Theorem 2** The test set generated by Algorithm 2 is combinatorial complete with respect to S and D.

Note:
1. A combinatorial complete test set covers all combinations of the operation conditions represented by the datamorphisms.
2. The test set generated by the algorithm may be not minimal.
3. A proof of the theorem can be found in the following paper:

1. ML Classification Models
2. Exploratory testing (ET) methodology
3. Datamorphic approach to automate ET
   a. Test system and completeness
   b. Test strategies
4. Application to testing feature-based ML classifiers
5. The uses of the information discovered by ET
Typical Classification Applications

Hong Zhu
AI Techniques to Develop Classifiers

- **Clustering**: (unsupervised learning)
  To find a way of partitioning data points into groups according to a similarity or a distance function

- **Classification**: (supervised learning)
  To find a function from a set of labelled data to classify the data into groups such that data of the same label are in the same class
Classifiers

A classifier (or a classification program) is a mapping $P: D \rightarrow G$ from the data space $D$ into a non-empty set of groups $G = \{l_1, ..., l_n\}$ (also called classes) such that $D = \bigcup_{l \in G} D_l$, where $D_l = \{x \in D | P(x) = l\}$, and $\forall x, y \in G. (x \neq y \Rightarrow D_x \cap D_y = \emptyset)$.

We assume that there is a distance function $\| \cdot \|: D^2 \rightarrow R^+$, such that

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

$\forall x, y \in D. (\|x, y\| \geq 0)$

$\forall x, y \in D. (\|x, y\| = \|y, x\|)$

$\forall x, y, z \in D. (\|x, y\| + \|y, z\| \geq \|x, z\|)$
Testing Classification Systems

- Traditional Approach: Category Partitioning Testing (also known as *domain analysis*)
  - **Focusing on the borders** between different classes,
    - Defined by the specification, or
    - As implemented by the code, or
    - A combination of the above
  - **Technique:**
    - For each class: selecting test cases on the borders and near-by to the borders
    - The number of test cases on or nearby to a border depends on the dimension of the data space
  - **Theory** (e.g. in the perturbation testing theory):
    - Test cases on the border and near-by to the borders can ensure no linear transformations of the border (e.g. border shift errors and rotate errors) under certain conditions on the border and data space.
Category Partitioning Test (Domain Analysis)

Class A

Border Shift Error

Class B

Border between Class A and B

Border Rotate Error
Can we borrow the Ideas of category partitioning test to ML?
Problems to Apply Partitioning Test to AI Applications

- The borders between classes are often unknown
  - No definition of the required border in the specification
  - Not easy to get the border as implemented by the ML model

- The data space and the borders are highly complicated
  - High dimensional
  - Non-numerical data

- The theory of domain analysis does not apply
  - The common errors in the application of AI technology may be not linear transformations (not border shift or rotate errors)
Examples of Errors in Machine Learning Models

The classifier:
$[0, 2\pi] \times [-1, 1] \rightarrow \{\text{red, blue, black}\}$

- Take 5000 random samples of the original classifier
- Apply various ML techniques to train ML models
- The result models are shown on the right:

17 July 2023
Tutorial on Datamorphic Testing
Exploratory Testing

“In exploratory testing, the tester interacts with the application and uses the information that the application provides to change the course of testing in order to explore the application’s functionality.”

[Whittaker, 2009]

“Simultaneously designing and executing tests to learn about the system, using your insights from the last experiment to inform the next.”

[Hendrickson, 2013]
### Confirmatory Testing

**Goal of Test:**
- Confirming or disproving the correctness with respect to a given specification
- Testing for verification and validation w.r.t. known requirements and specification

**Software under test:**
- As an entity with clear definition and specification
- Knowledge of the SUT is essential to perform testing

**Test cases:**
- Pre-scripted
- Independent from each other
- Quality criteria: to coverage all possibilities

### Exploratory Testing

**Goal of Test:**
- Discovering the functions and properties of the software
- Testing as experiments on the software
- To search for useful information

**Software under test:**
- As an entity unknown
- No knowledge of the SUT is assumed

**Test cases:**
- Generated or selected on the fly: using the result of the previous tests to guide the choice of the next
- **Quality criteria:** to maximise its effectiveness in the process of searching for useful information
Exploratory Testing: A Brief Review

- A primitive form in the practice of *manual testing* existed for a long time
- Most suitable for situations where *specification is not available* or not well defined
- Relatively recently identified by researchers to provide guidance to improve the effectiveness of manual testing of interactive software
  - Kane [1988] coined the term “exploratory test”
  - Whittaker [2009] recognised a defined (informally) strategies for GUI based testing
  - Many researchers conducted empirical studies of the factors that effect ET

Exploratory Testing of Classifiers

Goal:

- To discover the borders between classes as defined by the ML model under test
  - Borders are critical to understand the behaviour of a ML model
  - Values on borders are critical test cases for a ML model

Problems:

- How to represent borders?
- Can borders be discovered?
- If yes, how to discover borders?
- Is the discovery of borders cost efficient? Can it be automated?
- How to use borders?
Definition 1. (Pareto Front of Classification)
Let

- \( P : D \rightarrow G = \{l_1, \ldots, l_n\} \ (n > 0) \) be a classifier,
- \( \| \cdot, \cdot \| : D \times D \rightarrow \mathbb{R}^+ \) be a distance metric on \( D \), and
- \( \delta > 0 \) be any given real number.

A set \( \{(a_i, b_i) | a_i, b_i \in D, i = 1, \ldots, k\} \ (k > 0) \) of data pairs is a Pareto front of the classes according to \( P \) with respect to \( \| \cdot, \cdot \| \) and \( \delta \), if for all \( i = 1, \ldots, k \), \( P(a_i) \neq P(b_i) \) and \( \|a_i, b_i\| \leq \delta \).
Example: Pareto front

Classifier:

A Pareto front:
Essential Elements of Exploratory Testing

✓ Designing:
   It is concerned with identifying interesting things to vary and interesting ways in which to vary them so that the experiment can be better performed.

✓ Executing:
   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.
# Datamorphic Approach to Exploratory Test

<table>
<thead>
<tr>
<th>Essential Elements of ET</th>
<th>Datamorphic Approach to ET (**)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong>: Identifying interesting things to vary and interesting ways in which to vary them</td>
<td>Developing test morphisms to implement the ways in which to vary the test entities</td>
</tr>
<tr>
<td><strong>Executing</strong>: Executing a test as soon as you think of a test</td>
<td>Invoking the test executor on test cases</td>
</tr>
<tr>
<td><strong>Learning</strong>: Discovering how the software operates</td>
<td>Writing test code to analyse test results and present them in a format easy to digest by human beings</td>
</tr>
<tr>
<td><strong>Steering</strong>: Using knowledge gain from testing to suggest the next test with focus on most important information to discover</td>
<td>Formalising steering strategies in the form of algorithms that utilise test entities and morphisms as parameters</td>
</tr>
</tbody>
</table>


Exploratory Test System $\mathcal{T} = \langle \mathcal{E}, \mathcal{M} \rangle$

1. The set $\mathcal{M}$ of morphisms contains a test executer $\text{Exe}_P(x)$ that executes the program $P$ under test on a test case $x$ and receives the output of $P$; that is $\text{Exe}_P(x) = P(x)$. In the sequel, we will write $P(x)$ for $\text{Exe}_P(x)$ for the sake of simplicity.

2. There is a set $W \subseteq \mathcal{M}$ of unary datamorphisms defined on $D$. Informally, for each $w \in W$ and $x \in D$, $w(x)$, $w^2(x)$, $\cdots$, $w^n(x)$ generates a sequence of different 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 $m \in \mathcal{M}$ such that for all $x, y \in D$, $\text{dist}(x, z) < \text{dist}(x, y)$ and $\text{dist}(y, z) < \text{dist}(x, y)$, where $z = m(x, y) \in D$. Informally, the datamorphism $m$ calculates a point between $x$ and $y$. It is called the midpoint method.
Exploratory Test Systems

An *exploratory test system* is a test system $T = \langle E, M \rangle$ that $M$ has contains the following test morphisms.

- **A test executor** $Exe_P(x)$: through the test morphism the program $P$ under test are invoked on a test case $x$ and receives the output of $P$. That is, $Exe_P(x) = P(x)$.

- **A set of traversal methods**: a set $W \subseteq M$ of *unary datamorphisms* defined on $D$. For each $w \in W$ and $x \in D$, by repeatedly invoke the datamorphism $w$, i.e. $w(x), w^2(x), \cdots, w^n(x)$, we can generate a sequence of data points in $D$, where $w^1(x) = w(x), w^{n+1}(x) = w(w^n(x))$.

- **A midpoint method**: a *binary datamorphism* $m \in M$ such that
  \[
  \forall x, y \in D. (\| x, y \| \triangleright \delta_m \Rightarrow \| x, z \| < \| x, y \| \wedge \| y, z \| < \| x, y \|)
  \]
  where $z = m(x, y), \quad \delta_m = Min_{x \neq y \in D} \{\| x, y \| \}$.
Example: An Exploratory Test System

- The classifier under test:
  - **Input data space** $D$: $[0,2\pi] \times [-1,1]$
  - **Function**: classify into **red**, **blue** and **black**

- The distance metrics:
  
  $Eucl((x_1, x_2), (y_1, y_2)) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

- The datamorphisms:
  - upward($x$);
  - downward($x$);
  - leftward($x$);
  - rightward($x$);
  - mid($x$, $y$);

**Implementation of datamorphisms in Java**

```
@Datamorphism
cpyublic 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;
}
```
Completeness of Exploratory Test System

Definition 5. (*Completeness*)

An exploratory test system \( T = \langle E, M \rangle \) on data space \( D \) is *complete*, if for all \( a, b \in D \), there is a composition \( \varphi(x) \) of datamorphisms in \( M \) such that \( b = \varphi(a) \).

An exploratory test system \( T \) is *approximately complete*, if for all \( a, b \in D \) and every \( \delta > \delta_m \), there is a composition \( \varphi(x) \) of datamorphisms in \( M \) such that \( \| b, \varphi(a) \| \leq \delta \).

The completeness of an exploratory test system ensures that there will be no blind spot in the data space that cannot be explored.
Question:
Is there complete exploratory test system for ML classifiers?

Answer:
Yes, for feature-based classifiers, we can always construct a complete exploratory test system.
Definition 2. *(Feature Based Classifier)*

Let \( P : D \rightarrow G \) be a classification program. We say that \( P \) is a *feature-based classifier* if there is a natural number \( K \geq 1 \) such that \( D = D_1 \times \cdots \times D_K \), where for every \( i = 1, \ldots, K \), \( D_i \) is the set of values of a feature \( f_i \).

**Types of features:**

- A feature \( f_i \) is categorical, 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.
Datamorphisms for Continuous Numerical Features

- Two unary datamorphisms for each feature $f_i$ as the traversal methods

\[
up_i((x_1, \ldots, x_K)) = (x_1, \ldots, x_i + c_i, \ldots, x_K)
\]
\[
down_i((x_1, \ldots, x_K)) = (x_1, \ldots, x_i - c_i, \ldots, x_K)
\]

- A binary datamorphism $mid_E(x, y)$ as the midpoint method.

\[
mid_E((x_1, \ldots, x_K), (y_1, \ldots, y_K)) = \left(\frac{x_1+ y_1}{2}, \ldots, \frac{x_K + y_K}{2}\right)
\]

- The Euclidean distance on multi-dimensional real numbers.

\[
\| (x_1, \ldots, x_K), (y_1, \ldots, y_K) \| = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}
\]

There are many other ways to define distance metrics on real numbers.
Datamorphisms for Discrete Numerical Features

- Two unary datamorphisms for each discrete numerical feature $f_i$ as the traversal methods

\[
up_i(\langle x_1, \ldots, x_K \rangle) = \langle x_1, \ldots, x'_i, \ldots, x_K \rangle, \text{ where } x'_i = x_i + 1.
\]

\[
down_i(\langle x_1, \ldots, x_K \rangle) = \langle x_1, \ldots, x'_i, \ldots, x_K \rangle
\]

where $x'_i = x_i - 1$, if $D_i$ is the set of integers; otherwise

\[
x'_i = \begin{cases} 
    x_i - 1, & \text{if } x_i > 0 \\
    0, & \text{otherwise}
\end{cases}
\]

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

\[
mid_N(\langle x_1, \ldots, x_K \rangle, \langle y_1, \ldots, y_K \rangle) = \left( \left[ \frac{|x_1-y_1|}{2} \right], \ldots, \left[ \frac{|x_K-y_K|}{2} \right] \right)
\]

- The distance metric $\| \langle x_1, \ldots, x_K \rangle, \langle y_1, \ldots, y_K \rangle \|_N = \sum_{i=1}^{K} |y_i - x_i|$
Datamorphisms for Categorical Features

- Two unary datamorphisms as the traversal methods for each categorical feature $f_i$

  $$up_i(\langle x_1, \cdots, x_K \rangle) = \langle x_1, \cdots, x'_i, \cdots, x_K \rangle,$$
  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}
  \\
  down_i(\langle x_1, \cdots, x_K \rangle) = \langle x_1, \cdots, x'_i, \cdots, x_K \rangle,$$
  where $x'_i = \begin{cases} v_{i,j-1} & \text{if } x'_i = v_{i,j} \text{ and } j > 1 \\ v_{i,1} & \text{if } x'_i = v_{i,1} \end{cases}$

- A binary datamorphism $mid_D(x, y)$ as the midpoint method

  $$mid_D(x, y) = \langle z_1, \cdots, z_K \rangle,$$
  where $z_i = \begin{cases} x_i & \text{if } x_i = y_i \\ x_i & \text{if } x_i \neq y_i \text{ and } x_i \text{ is an odd-indexed element in } \Delta(x, y) \\ y_i & \text{if } x_i \neq y_i \text{ and } x_i \text{ is an even-indexed element in } \Delta(x, y) \end{cases}$

- The distance between $x$ and $y$, written $\| x, y \|_D$, is defined as the number of elements in $x$ and $y$ that are different.
Exploratory Test System for Feature-based Classifiers

Let $x = \langle d_1, \cdots, d_u, n_1, \cdots, n_v, r_1, \cdots, r_w \rangle \in D$.

$x_D = \langle d_1, \cdots, d_u \rangle$, $x_N = \langle n_1, \cdots, n_v \rangle$, and $x_E = \langle r_1, \cdots, r_w \rangle$.

Define $\oplus$ such that $x = x_D \oplus x_N \oplus x_E$.

- Two unary datamorphisms $up_i$ and $down_i$ for each feature $f_i$
  - Definition of the datamorphisms depends on the type of feature; see previous slides

- A binary datamorphism $mid_H(x, x')$ as the midpoint method
  
  \[
  mid_H(x, x') = mid_D(x_D, x'_D) \oplus mid_N(x_N, x'_N) \oplus mid_E(x_E, x'_E)
  \]

- The distance function $\| \cdot, \cdot \|_H : D \times D \rightarrow R^+$ as follows.
  
  \[
  \| x, x' \|_H = \| x_D, x'_D \|_D + \| x_N, x'_N \|_N + \| x_E, x'_E \|_E.
  \]

**Theorem.** The above set of datamorphisms and the distance metrics $\| \cdot, \cdot \|_N$ together satisfy the requirements of exploratory test systems on datamorphisms, and it is approximately complete.
1. Definitions of the strategies as algorithms
   a) Random target
   b) Directed walk
   c) Random walk
2. Proofs of the correctness of the algorithms
3. Evaluation of performance of the algorithms
Strategy 1: Random Target

Select a number of pairs of points in the space $D$ at random, if a pair of points are in different class, using the midpoint method repeatedly to find a pair border points between them.

**Stage 1:** Select two points 1 and 2 at random. Success and progress to Stage 2, if the points are in different classes; otherwise fail and terminate.

**Stage 2:** Repeatedly taking the midpoint of the last two points in different classes for a number of times to ensure the distance between the last two points is smaller than the target distance of the pareto
Algorithm 1 (Random Target Strategy)

Input:
- \textit{testSet}: Test Pool;
- \textit{steps}: Integer;
- \textit{mid}(x, y): Binary datamorphism;

Output:
- \textit{a, b}: Test Case;

Begin
1: Select two different test cases \( x \) and \( y \) in \textit{testSet} at random;
2: Execute program \( P \) on test cases \( x \) and \( y \);
3: Check if a pair of Pareto front exits between \( x \) to \( y \):
   \textbf{if} \ (x.output = y.output) \textbf{then} \textbf{return} \ (null, null) \textbf{end if}
4: Refinement:
   \textbf{for} \ i \leftarrow 1 \ \textbf{to} \ \textit{steps} \ \textbf{do}
      \( z = \text{mid}(x, y) \);
      \textbf{if} \ (x.output \neq z.output) \textbf{then} \ y = z \textbf{else} \ x = z \textbf{end if}
   \textbf{end for};
   \( a = x; \ b = y \);
   \textbf{return} \ (a, b);
End
Correctness of The Random Target Algorithm

Assume that the exploratory test system has the following properties.

1. There is a constant $c > 1$ such that

\[
\forall x, y \in D. \left( \frac{\text{Max}\{\text{dist}(x, z), \text{dist}(z, y)\}}{\text{dist}(x, y)} \right) \leq \frac{1}{c},
\]

where $z = \text{mid}(x, y)$.

2. There is a constant $d_m > 0$ such that

\[
\forall x, y \in D. (\text{dist}(x, y) \leq d_m).
\]

**Theorem 1.** If $RT(n) = \langle a, b \rangle \neq \langle \text{null}, \text{null} \rangle$, then $\langle a, b \rangle$ is a pair of Pareto front according to $P$ with respect to $\text{dist}$ and $\delta$, if $d_m / c^n < \delta$. 
Example: Execution of The Random Target Strategy

- 1000 random pairs selected from 300 random test cases
- Number of steps: 20
- Number of pairs in the generated pareto front: 641
- Success rate: 64.1%
- The distance between each pair in the Pareto front: \[ \delta \leq \frac{d_m}{c^{20}} = \frac{\sqrt{\pi^2 + 1}}{2^{19}}. \]
Strategy 2: Directed Walk

Select a number of points in $D$ at random as the start points. From each point, use a walking method to traverse in one direction until find a point in different class, then find the border points between them using the midpoint methods repeatedly.

Stage 1: Start from one point $\mathbf{1}$ in the data space. Repeatedly using a given walking method to walk in one direction until find a point (point $\mathbf{5}$ in the figure) of different class. Fail and terminate, if repeated more than a set number of walking steps but still find no point in a different class.

Stage 2: Repeatedly taking the midpoint of the last two points in different classes for a number of times to ensure the distance between the last two points is smaller than the required distance of the Pareto.
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 ← 1 to walkDistance do
    y = d(x);
    Execute software on test case y;
    if (x.output ≠ y.output) then
      found = true; break;
    else x = y;
  end if
  end for
  4: //Check if a Pareto front can be found
  if (¬found) then return ⟨null, null⟩;
  end if
  5: //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
Correctness of The Directed Walk Strategy

Assume that the exploratory test system has the following properties

1. There is a constant $c > 1$ such that
   \[
   \forall x, y \in D. \left( \frac{\text{Max}\{\text{dist}(x, z), \text{dist}(z, y)\}}{\text{dist}(x, y)} \right) \leq 1/c,
   \]
   where $z = \text{mid}(x, y)$.

2. There is a constant $d_s > 0$ such that
   \[
   \forall x \in D. (\text{dist}(x, d(x)) \leq d_s).
   \]
   where $d_s$ is called the step size of the traversal method $d(x)$.

\textbf{Theorem 2.} If $\text{DW}(m, n) = \langle a, b \rangle \neq \langle \text{null}, \text{null} \rangle$, then, $\langle a, b \rangle$ is a pair in the Pareto front according to $P$ with respect to dist and $\delta$, if $d_s / c^n < \delta$, where $n$ is the number of steps.
Example: Execution of The Directed Walk Strategy

- 1000 start points selected at random; Walk direction: upward
- Walking distance: 20 steps; Number of refinement steps: 20
- Number of pairs in the generated pareto front: 161
- Success rate: 16.1%
- Distance between points in each pair: \( \delta \leq \frac{d_s}{c^{20}} = 0.2 \times \frac{1}{2^{20}} \).
Strategy 3: Random Walk

Select a number of points in $D$ at random as the start points. From each point, use a number of walking method to walk randomly (each step choice a walking method at random), until a point of different class is find, and then find a pair of border points using the midpoint method repeatedly.

**Stage 1:** Start from one point $1$, repeatedly using a walking method selected at random until find a point of different class (point $5$ in the figure). Fail, if repeated more than a set number of walking steps but still find no point in a different class.

**Stage 2:** Repeatedly taking the midpoint of the last two points in different classes for a number of times to ensure that the distance between the last two points is smaller than the target distance of the pareto.
**Algorithm 3 (Random Walk)**

**Input:**
- `testSet`: Test Set;
- `walkingDistance`: Integer;
- `steps`: Integer;
- `d_1(x)`, `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 ← 1 to `walkingDistance` do**
  - Get a random integer `r` in the range `[1, k]`
  - `y = d_r(x);`

End

Execute program `P` on test case `y`;
if `(x.output ≠ y.output)` then
  - `found = true; break;`
else `x=y;`
end if
end for
if `(¬found)` 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⟩;`
Correctness of The Random Walk Algorithm

Assume that the exploratory test system has the following properties

(1) There is a constant $c > 1$ such that

$$\forall x, y \in D. \left( \frac{\text{Max}\{\text{dist}(x, z), \text{dist}(z, y)\}}{\text{dist}(x, y)} \right) \leq 1/c,$$

where $z = \text{mid}(x, y)$.

(2) There is a constant $d_s > 0$ such that

$$\forall x \in D. \forall d_i \in W. (\text{dist}(x, d_i(x)) \leq d_{sm}). \quad (6)$$

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 3.** If $RW(m, n) = \langle a, b \rangle \neq \langle \text{null}, \text{null} \rangle$, then, $\langle a, b \rangle$ is a pair of Pareto front according to $P$ with respect to $\text{dist}$ and $\delta$, if $d_{sm}/c^n < \delta$, where $n$ is the steps.
Example: Execution of The Random Walk Strategy

- 1000 random walks with 300 starting points selected at random
- Walking distance: 20 steps; Number of refinement steps: 20
- Walk directions: upward, downward, leftward, rightward
- Number of pairs in the generated pareto front: 805
- Success rate: 80.5%
- Distance between points in each pair: \( \delta \leq \frac{d_s}{c^{20}} = 0.2 \times \frac{1}{2^{20}} \).
Evaluation of the Strategies

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?

We measure the cost using the average number of test executions of the classifier for discovering each pair in the Pareto front.

Capability is the probability of a test strategy returning a Pareto front pair when executed.

\[ E_m = \frac{||PF||}{W} \]

Cost is the amount of computational resources needed to find a pair in a Pareto front.

\[ Time(W) = E_m \times C_m \times W \times s_m \]

Time needed to take \( W \) walks

Time needed to invoke the model \( m \) once

Cost = Average number of invocations of model \( m \) for each pair in PF

Size of Pareto front of model \( m \)

Capacity of testing model \( m \)

Number of walks (executions of the strategy)
Subjects of The Empirical Evaluations (1)

Controlled Experiment with 10 manually coded classifiers

- **Input domain:**
  - Two-dimensional real numbers in the range of $[0, 2\pi] \times [-1, 1]$.

- **Output classes:**
  - {Red, Blue, Black}
Subjects of The Empirical Evaluations (2)

Case study with ML models built from real datasets

- **Red Wine Quality**
  
  Quality of red varieties of the Portuguese “Vinho Verde” wine (Cortez et al., 2009).

- **Mushroom Edibility**
  

- **Bank Churners**
  
  Data of creditcard customers used to predict churners, who are bank customers who leave the credit card service.

### Table 2: Summary of Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Records</th>
<th>Classes</th>
<th>DF</th>
<th>NF</th>
<th>CF</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Wine Quality</td>
<td>1599</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Mushroom Edibility</td>
<td>8124</td>
<td>2</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>Bank Churners</td>
<td>10127</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>3</td>
<td>19</td>
</tr>
</tbody>
</table>
## Machine Learning Models Constructed for Each Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>Logistic Regression</td>
<td>Trained on whole data set</td>
</tr>
<tr>
<td>LR2</td>
<td>Logistic Regression</td>
<td>Used train-test 90-10 split</td>
</tr>
<tr>
<td>KNN</td>
<td>K-Nearest Neighbors</td>
<td>Trained on whole data set</td>
</tr>
<tr>
<td>KNN2</td>
<td>K-Nearest Neighbors</td>
<td>Used train-test 90-10 split</td>
</tr>
<tr>
<td>DT</td>
<td>Decision Tree</td>
<td>Trained on whole data set</td>
</tr>
<tr>
<td>DT2</td>
<td>Decision Tree</td>
<td>Used train-test 90-10 split</td>
</tr>
<tr>
<td>NB</td>
<td>Naive Bayes</td>
<td>Trained on whole data set</td>
</tr>
<tr>
<td>NB2</td>
<td>Naive Bayes</td>
<td>Used train-test 90-10 split</td>
</tr>
<tr>
<td>SVM</td>
<td>Support vector machine</td>
<td>Trained on whole data set</td>
</tr>
<tr>
<td>SVM2</td>
<td>Support vector machine</td>
<td>Used train-test 90-10 split</td>
</tr>
<tr>
<td>SV</td>
<td>Ensemble via Soft Voting</td>
<td>Trained on whole data set; LR+KNN+DT</td>
</tr>
<tr>
<td>SV2</td>
<td>Ensemble via Soft Voting</td>
<td>Used train-test 90-10 split; LR+KNN+DT</td>
</tr>
<tr>
<td>HV</td>
<td>Ensemble via Hard Voting</td>
<td>Trained on whole data set; LR+KNN+DT</td>
</tr>
<tr>
<td>HV2</td>
<td>Ensemble via Hard Voting</td>
<td>Used train-test 90-10 split; LR+KNN+DT</td>
</tr>
<tr>
<td>Stack1</td>
<td>Ensemble via Stacking</td>
<td>Used train-test 90-10 split; KNN as Meta; LR2+KNN2+DT2+HV2</td>
</tr>
<tr>
<td>Stack3</td>
<td>Ensemble via Stacking</td>
<td>Used train-test 90-10 split; LR as Meta; KNN2+DT+SV2+HV2</td>
</tr>
</tbody>
</table>

A total of 48 machine learning models are built and used in the case study.
Experiment Process

- For each subject application, three exploration strategies are executed with various parameters.
- For each setting of parameters, the exploration strategy algorithm is executed repeatedly for 10 times.
- For each execution of the strategy on each model, the number of invocations of the model under test and the size of Pareto front generated are recorded.
- The average of the data collected in 10 executions is used to analyse the results.

Used the testing tool Morphy
- The exploratory test system are written in Java.
- Morphy test scripts are written to automatically conducted the experiments.
- Tests are executed using Morphy.

Morphy tool, test code, test scripts and data are on GitHub:
https://github.com/hongzhu6129/ExploratoryTestAI.git
Main Results: Coded Classifiers

<table>
<thead>
<tr>
<th>Subject</th>
<th>Directed Walk</th>
<th></th>
<th></th>
<th></th>
<th>Random Target</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Cap</td>
<td>Cost</td>
<td>Cap</td>
<td>Cost</td>
<td>Cap</td>
<td></td>
</tr>
<tr>
<td>Box 1</td>
<td>323.45</td>
<td>50.53</td>
<td>52.46</td>
<td>20.72</td>
<td>11.49</td>
<td>12.69</td>
<td></td>
</tr>
<tr>
<td>Box 2</td>
<td>93.85</td>
<td>50.53</td>
<td>22.83</td>
<td>51.59</td>
<td>10.38</td>
<td>50.53</td>
<td></td>
</tr>
<tr>
<td>Circle 1</td>
<td>247.32</td>
<td>20.67</td>
<td>42.59</td>
<td>26.03</td>
<td>10.93</td>
<td>21.49</td>
<td></td>
</tr>
<tr>
<td>Circle 2</td>
<td>105.82</td>
<td>47.32</td>
<td>25.50</td>
<td>46.01</td>
<td>10.41</td>
<td>48.31</td>
<td></td>
</tr>
<tr>
<td>Line 1</td>
<td>105.82</td>
<td>49.15</td>
<td>29.02</td>
<td>40.13</td>
<td>10.41</td>
<td>48.25</td>
<td></td>
</tr>
<tr>
<td>Line 2</td>
<td>55.76</td>
<td>58.03</td>
<td>23.94</td>
<td>48.56</td>
<td>10.33</td>
<td>58.40</td>
<td></td>
</tr>
<tr>
<td>Sin 1</td>
<td>122.35</td>
<td>50.10</td>
<td>20.65</td>
<td>45.51</td>
<td>10.38</td>
<td>49.76</td>
<td></td>
</tr>
<tr>
<td>Sin 2</td>
<td>64.75</td>
<td>62.34</td>
<td>26.03</td>
<td>60.54</td>
<td>10.31</td>
<td>61.76</td>
<td></td>
</tr>
<tr>
<td>Triangle 1</td>
<td>370.38</td>
<td>7.62</td>
<td>66.79</td>
<td>16.06</td>
<td>12.46</td>
<td>8.33</td>
<td></td>
</tr>
<tr>
<td>Triangle 2</td>
<td>93.19</td>
<td>46.96</td>
<td>23.98</td>
<td>49.08</td>
<td>10.41</td>
<td>47.01</td>
<td></td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td><strong>158.27</strong></td>
<td><strong>44.32</strong></td>
<td><strong>33.38</strong></td>
<td><strong>40.46</strong></td>
<td><strong>10.75</strong></td>
<td><strong>40.65</strong></td>
<td></td>
</tr>
</tbody>
</table>
Main Results: Coded Classifiers

### Cost

<table>
<thead>
<tr>
<th></th>
<th>Box 1</th>
<th>Box 2</th>
<th>Circle 1</th>
<th>Circle 2</th>
<th>Line 1</th>
<th>Line 2</th>
<th>Sin 1</th>
<th>Sin 2</th>
<th>Triangle 1</th>
<th>Triangle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Walk</td>
<td>40.46</td>
<td>40.46</td>
<td>10.75</td>
<td>10.75</td>
<td>20.65</td>
<td>20.65</td>
<td>8.95</td>
<td>8.95</td>
<td>11.78</td>
<td>11.78</td>
</tr>
<tr>
<td>Random Target</td>
<td>33.38</td>
<td>33.38</td>
<td>12.46</td>
<td>12.46</td>
<td>26.03</td>
<td>26.03</td>
<td>10.17</td>
<td>10.17</td>
<td>10.17</td>
<td>10.17</td>
</tr>
<tr>
<td>Directed Walk</td>
<td>28.15</td>
<td>28.15</td>
<td>10.31</td>
<td>10.31</td>
<td>20.67</td>
<td>20.67</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>

### Capability

<table>
<thead>
<tr>
<th></th>
<th>Box 1</th>
<th>Box 2</th>
<th>Circle 1</th>
<th>Circle 2</th>
<th>Line 1</th>
<th>Line 2</th>
<th>Sin 1</th>
<th>Sin 2</th>
<th>Triangle 1</th>
<th>Triangle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Walk</td>
<td>370.38</td>
<td>370.38</td>
<td>40.65</td>
<td>40.65</td>
<td>47.01</td>
<td>47.01</td>
<td>40.13</td>
<td>40.13</td>
<td>40.13</td>
<td>40.13</td>
</tr>
<tr>
<td>Random Target</td>
<td>23.98</td>
<td>23.98</td>
<td>49.08</td>
<td>49.08</td>
<td>48.31</td>
<td>48.31</td>
<td>46.01</td>
<td>46.01</td>
<td>46.01</td>
<td>46.01</td>
</tr>
<tr>
<td>Directed Walk</td>
<td>50.53</td>
<td>50.53</td>
<td>20.67</td>
<td>20.67</td>
<td>12.69</td>
<td>12.69</td>
<td>50.53</td>
<td>50.53</td>
<td>50.53</td>
<td>50.53</td>
</tr>
</tbody>
</table>

Figure 12: Test Cost and Capability on Subject Programs
Main Results: Real Machine Learning Models

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Subject</th>
<th>Cost</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>Directed Walk</td>
<td>Red Wine Quality</td>
<td>63.03</td>
<td>14.12</td>
</tr>
<tr>
<td></td>
<td>Mushroom Edibility</td>
<td>32.63</td>
<td>18.90</td>
</tr>
<tr>
<td></td>
<td>Bank Churners</td>
<td>35.56</td>
<td>14.07</td>
</tr>
<tr>
<td>Random Target</td>
<td>Red Wine Quality</td>
<td>33.14</td>
<td>11.47</td>
</tr>
<tr>
<td></td>
<td>Mushroom Edibility</td>
<td>12.61</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>Bank Churners</td>
<td>18.81</td>
<td>12.40</td>
</tr>
<tr>
<td>Random Walk</td>
<td>Red Wine Quality</td>
<td>40.87</td>
<td>14.31</td>
</tr>
<tr>
<td></td>
<td>Mushroom Edibility</td>
<td>488.50</td>
<td>21.42</td>
</tr>
<tr>
<td></td>
<td>Bank Churners</td>
<td>30.34</td>
<td>8.10</td>
</tr>
</tbody>
</table>
Main Results: Real Machine Learning Models

### Capability

![Capability Graph](image1)

### Cost

![Cost Graph](image2)

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

17 July 2023
Main Findings 1: Answers to Research Questions

- **RQ1**: The strategies are capable of discovering borders between subdomains.
  - The overall average of the capabilities of all three strategies: 34.48%.
  - Directed walk: 21.86%
  - Random target: 31.47%
  - Random walk: 50.10%

- **RQ2**: Applying exploratory strategies is cost efficient for discovering borders between classes.
  - The overall average cost: 26.32 (of three strategies over all subjects)
  - The best cost: 6.23. (achieved in the testing of mushroom edibility models using the random target strategy)
  - The worst cost: 92.01 (observed also when testing mushroom edibility but using the random walk strategy).
Main Findings 2: Factors that Determine Capability

- Directed walk strategy:
  The probability that there is a border between two subdomains in the right direction from a test case and within the walking distance

- Random target strategy:
  The probability that two random test cases fall in two different subdomains

- Random walk strategy:
  The probability that there is a border nearby to a randomly selected test case
Main Findings 3: Properties of The Strategies

The data of the case study of real machine learning models are consistent with the data of the controlled experiments on both capability and cost of the strategies.

- The capability and cost are invariant in the number of walks.
  - Both cost and capability are constants that only vary with the model under test.

- The dimensions of the input data spaces of the real-world examples are significantly larger than those coded classifiers.
  - The strategies are scalable to high dimensional data spaces.
Uses of Pareto Front: 1. Measuring Error Extent

Uses of Pareto Front: 2. Explanation

Deep Neural Network (DNN)
Uses of Pareto Front: 3. Visualisation

- Original Coded Classifier
- Deep Neural Network (DNN)
- Decision Tree (DT)
- Hard Voting of LR, KNN and DT (HV)
- K-Nearest Neighbour (KNN)
- Logistic Regression (LR)
Pareto Front of Box2 Models (Generated via 5K Random Walks)

- Original Coded Classifier
- Deep Neural Network (DNN)
- Decision Tree (DT)
- Hard Voting of LR, KNN and DT (HV)
- K-Nearest Neighbour (KNN)
- Logistic Regression (LR)
- Naïve Bayes (NB)
- Soft Voting of LR, KNN and DT (SV)
- Supporting Vector Machine (SVM)
- Stacking KNN over LR, DT and HV (Stack)
Uses 4: Testing ML Model’s Robustness

The notion of Pareto front of classification generalises the notion of adversarial examples.

Pareto front links from adversarial examples to critical decision region

- **Distance metric**: number of pixels that are different;
- **Tolerable error**: 1

The critical pixel that on which the ML model decides the classification

Decision region = the set of critical pixels

The difference bwt the pair is on only one pixel
Scenario-based Exploratory Functional Testing

CISOSE 2023 Invited Track
Session 5, 17th July 2023 (Monday) 14:00pm
(Auditorium)

Thank You