Verification and Validation

Yiding Li
McMaster University

1 Introduction

Verification and validation (**VnV**) ensures that the project sufficiently satisfies its requirements. If the project fails to satisfy its requirements, then VnV discusses and documents the extent of this failure.

VnV splits into **verification** and **validation**. Verification verifies the software against a set of requirements; validation considers the necessity and feasibility of requirements.

1.1 Role and Responsibilities

This document is the **verification and validation plan (VnV plan)**, which documents the planning and instructs the implementation of VnV procedures. The VnV plan starts after the requirements stage and before the design stage, then undergoes iterative refinements as development continues.

This document references the following design artefacts:

- SRS describes requirements specifications that must be tested.

1.2 Limitations of Testing

Testing cannot ensure correctness, only that the program performs correctly for the tested test cases. Testing only builds confidence that the software is correct. Techniques (such as unit testing) and metrics (such as coverage metrics) improve the effectiveness of tests at building performance.

1.3 Characteristics of Intended Readers

This document is designed for readers with the following roles and characteristics. Readers with these characteristics should be able to understand this document.

Developers design and implement Agolearn test cases following this document. Developers have the following characteristics:

- Understanding of modular testing
- Familiarity with evolutionary learning comparable to a graduate course on evolutionary learning

Users consult test cases to understand Agolearn and its capabilities. Users have the following characteristics:

- Understanding of modular software design
- Familiarity with evolutionary learning comparable to a graduate course on evolutionary learning

2 Scope of Tests

This section discusses what tests do and do not cover.

2.1 Assumptions

Validation and verification procedures make the following assumptions:

- **Locality**. The library is always available and exists locally. Access to the internet is unavailable.

- **The environment is abstract**. System-dependent resources, such as environment variables and system calls, are not explicitly available.
- **Direct access**. Test primitives can directly invoke the tested interface.
- **Resources are abundant**. Tests can use as much computational resource as necessary.

2.2 Ignored Categories

The following properties are either infeasible or impossible to test. Measure these properties, but do not write test cases for these measurements.

- Performance. The software is not performance sensitive.

2.3 Machinery

The following subsections describe the machinery of testing.

- **Oracles and Pseudo-Oracles** are programs that behave similarly to this project. They provide
- **Primitives** are functions that facilitate the testing process.

2.3.1 Constants

The following table describes testing constants in the following types:

- **Ground truths** are constants known before the fact. These constants describe the reality that is captured by the project.
- **Test Constants** are constants only used in test cases.
- **Error Margins** are special test constants that describe the leniency of test cases. A test case passes if the difference is within the given error margin.

TABLE 1: GROUD TRUTHS

TABLE 2: TEST CONSTANTS

TABLE 3: MARGINS OF ERROR

δ_F	Error margin for floating-point comparisons

2.3.2 Pseudo-Oracles

Pyvolution - a modular evolutionar learning framework

2.3.3 Primitives

The assert_epsilon primitive compares floating-point numbers, accounting for floating-point errors.

```
assert_epsilon: ℝ × ℝ × ℝ → {⊤, ⊥}
¶ assert_epsilon(a, b, ε)
    if (|a − b| < ε) do
        return ⊤
    else
        return ⊥
    end if
end ¶
```

2.4 Test Plan

2.4.1 Invariants

- The size of a population should adhere to operators.
- The size of a population should adhere to operators.

2.4.2 Constraints

None at the moment

2.4.3 Pre- and Post-Conditions

PC 1: The population size follows variators and mutators.

2.4.4 Functional Requirements

FR 1: Type check input values

FR 2: Interchangeability of evolutionary operators of the same kind

FR 3: Each iteration produces a genome

FR 4: Subsequent genomes should not have lower fitness.

2.4.5 Nonfunctional Requirements

NFR 1: Usability: the interface should be inspected and approved by a university professor who specializes in evolutionary learning.

NFR 2: Usability: design artefacts should be inspected and approved by a university professor who specializes in evolutionary learning.

NFR 3: Maintainability: all functions should use type hints. All public documents should be documented.

NFR 4: Portability: Test cases should pass on a version of Microsoft Windows 11.

2.4.6 Coverage

Test cases should achieve 100% function coverage. Record then document statement coverage.

2.5 Test Structure

2.5.1 Methods
2.5.2 Modules
2.5.3 Dependencies

2.6 Test Items

2.6.1 Requirements

The fitness of the population should not decrease.

2.6.2 Variator

Degenerate cases

If the variator is identity, then the population should not contain new solutions.

If the variator is random, then no improvement should occur throughout generations.

2.6.3 Evaluator

Degenerate cases

If the evaluator is random, then no improvement should occur throughout generations.

2.6.4 Selector

Degenerate cases

If the selector is random, then no improvement should occur throughout generations.