Project Title: System Verification and Validation Plan for Agolearn

Team 1, Agonaught(s)
Yiding Li

March 11, 2024

Revision History

Date	Version	Notes
2024-02-29	1.0	First Draft
2024-03-11	1.1	Transcription to latex

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	description
SRS	Software requirements specification
VNV	Verification and validation
Module guide	
MIS	Module interface specification

This document ... [provide an introductory blurb and roadmap of the Verification and Validation plan —SS]

2 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 soft-ware against a set of requirements; validation considers the necessity and feasibility of requirements.

This section divides into the following subsections:

1. **Objectives** discusses the definition of verification and validation and the purpose of this document
2. **Characteristics of Intended Readers**— discusses characteristics required to understand this document
3. **Relevant Documentation** lists documents that are referenced in this document.

2.1 Objectives

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.

2.1.1 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.

2.2 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 Agolearnand its capabilities. Users have the following characteristics:

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

2.3 Relevant Documentation

This document references the following documents:

- SRS.pdf: Requirement specifications

3 Scope of Tests

This section discusses what tests do and do not cover. This section includes the following subsections:

1. **Assumptions** lists conditions that are assumed to be true by all test cases
2. **Ignored Categories** lists conditions that are not tested
3. **Machinery** lists machinery that are used to describe test cases

3.1 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.

3.2 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.

3.3 Machinery

The following subsections describe the machinery of testing.

1. **Constants** are immutable values.
2. **Pseudo-Oracles** are reference implementations that behave similarly to Agolearn.
3. **Interfaces** lists objects that are used in testing, and describes how to interact with these objects.
4. **Primitives** are functions that facilitate the testing process.

3.3.1 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: Ground Truths

Symbol	**Description**
Placeholder	what

Table 2: Test Constants

Symbol	**Description**
Placeholder	what

Table 3: Error Margins

Symbol	**Description**
δ_F	Error margin for floating-point comparisons

3.3.2 Pseudo-Oracles

- **Pyvolution**: A modular evolutionary learning framework

3.3.3 Interfaces

A genome describes a solution representation. The fitness of a genome positively correlates to its suitability as a solution.

[TODO: Find better names for 'arity']

Table 4: Genome

Name	Genome	
Type	Data Structure	
Attributes	fitness : Real	Quality of solution

Table 5: Population Factory

Name	population_factory
Type	Set[Genome] $\rightarrow$ Population

Table 6: Population

Name	Population	
Type	Data Structure	
Attributes	size : $\text{Int}_{\geq 0}$	Number of genomes in population

Table 7: Variator

Name	Variator	
Type	Population $\rightarrow$ Population	
Attributes	arity : $\text{Int}_{\geq 0}$	Input arity
	coarity : $\text{Int}_{\geq 0}$	Output arity
	residual : $\text{Int}_{\geq 0}$	Extra children

Table 8: Selector

Name	Selector	
Type	Population $\rightarrow$ Population	
Attributes	outsize : $\text{Int}_{\geq 0}$	Size of the output population

Table 9: Evaluator

Name	Evaluator
Type	Genome $\rightarrow$ Real

3.3.4 Primitives

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

```
assert_epsilon : $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \{\bot, \top\}$
assert_epsilon($a, b, \varepsilon$):
if $|a - b| < \varepsilon$ then
    return $\top$
else
    return $\bot$
end
```

The type primitive returns the mathematical type of its argument.

```
type : Any $\rightarrow$ Type
type($o$):
return the type of $o$
```

The best_fit primitive returns a member of a population with the highest fitness

```
best_fit : Population $\rightarrow$ Genome
best_fit($G$):
return $\text{argmax}_{g \in G}\{g.\text{fitness}\}$
```

3.4 Test Plan

3.4.1 Verification and Validation Team

The VnV team includes verifiers and validators. **Verifiers**: The following people review the implementation of this project.

Name	Role
Yiding Li	Designer and implementor of test cases
Stephen Kelly	Domain expert and usability reviewer

Validators: The following people review design artefacts.

Name	Role
Yiding Li	Author of development artefacts
Spencer Smith	Course instructor, reviewer
Fasil Cheema	Domain expert and primary reviewer
Fatemah	SRS reviewer
Tanya Djavaher-pou	VnV reviewer
Phil Du	MG + MIS reviewer

3.5 Test Plan

This section describes elements that should be tested, and how to test them. These elements are as follows:

- **Conditions**: preconditions, postconditions, and invariants
- **Functional Requirements**: functional requirements sourced from SRS.pdf
- **Nonfunctional Requirements**: nonfunctional requirements sourced from SRS.pdf
- **Coverage**: coverage criteria that should be met by test cases.

- **Traceability** Information: Reference from test cases to requirements.

3.5.1 Conditions

C1: The size of a population follows its variators.

Table 10: T1: Test Plan for C1

Subject	A variator
Control	Manual
Initial State	A population P is initialized and available A variator V is initialized and available
Input	A population P
Output	A population P' whose size is $(P/(V.\text{arity})) \cdot V.\text{coarity} + \text{extra}$

C2: The size of a population should adhere to selectors.

Table 11: T2: Test Plan for C2

Subject	A selector
Control	Manual
Initial State	A population P is initialized and available A variator V is initialized and available
Input	A population P
Output	A population P' whose size is $\max\{S.\text{outsize}, P.\text{size}\}$

3.5.2 Functional Requirements

FR1: Type check input values

Table 12: T3: Test Plan for FR1

Subject	population_factory
Control	Manual
Initial State	A set of genomes G is available
Input	A set of genomes G

Output	**if** type$(G) isSet[Real \rightarrow Real]$ $ortype(G)isSet[(A \rightarrow B) \rightarrow Real]$ **then**
	   do nothing
	 else
	   raise error
	 end

FR2: Interchangeability of evolutionary operators of the same kind

T4: Each individual selector, variator, and evaluator must pass all test cases for selector, variators, or evaluators respectively.

FR 3: Each iteration produces a population

T5: Static validation: each iteration results in a population.

FR 4: Subsequent genomes should not have lower fitness.

Table 13: T6: Test Plan for FR4

Subject	System
Control	Manual
Initial State	A population P is available
A variator V	
A selector S	
A evaluator E	
Input	A population P
Output	Apply E item-wise to $S(V(P))$. The highest-fitness item in the output P' should have a higher fitness than the highest-fitness item in the previous population P.

3.6 Non-Functional Requirements

FR 3: Each iteration produces a population

T5: Static validation: each iteration results in a population.

NFR1: Usability

T7: The interface should be inspected and approved by Kelly.

NFR 2: Understandability

T8: Design artefacts should be inspected and approved by members of the validation team.

NFR 3: Maintainability

T9: Static verification: All functions should use type hints. All arguments of all public documents should be documented.

NFR 4: Portability

T10: Test cases should pass on a version of Microsoft Windows 11.

3.6.1 Coverage

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