Verification and Validation Report: Re-ProtGNN

Yuanqi Xue

April 19, 2025

1 Revision History

Date	Version	Notes
April 15	1.0	Initial Draft
April 18	1.1	Final Version

2 Symbols, Abbreviations and Acronyms

symbol	description
T	Test
M	Module
R	Requirement
SRS	Software Requirements Specification
ProtGNN	Prototype-based Graph Neural Network
Re-ProtGNN	Re-implementation of the ProtGNN model

Contents

List of Tables

This document presents the results of the VnV Plan (**?**) for Re-ProtGNN. The requirements and modules were tested and verified through automated unit tests, system tests, and usability evaluations. GitHub Actions was used for continuous integration to ensure correctness throughout development.

3 Functional Requirements Evaluation

3.1 R1: Dataset Input Validity

The system tests T1, T2, and T3 confirmed that the system correctly handles valid and invalid input files:

- T1 raised FileNotFoundError when raw files were missing.
- T2 raised ValueError for non-numeric data.
- T3 verified successful loading of a valid dataset (MUTAG) (**?**).

All tests passed as expected.

3.2 R2: Training Loss Convergence

System test T2 (Loss Convergence Test) confirmed that the training loss decreased across epochs. The test validated:

- The final loss was lower than the initial loss.
- The model trained on MUTAG dataset (**?**) using the main training function.

This test indirectly verified the correctness of the top-level `train()` function.

3.3 R3: Inference Output Consistency

System test T3 confirmed that the trained model achieved above 50% classification accuracy (i.e, better than random guessing and performed mearningful learning) and returned prediction logits and probabilities in the correct shape. The test passed all assertions on value shape, alignment, and threshold.

4 Nonfunctional Requirements Evaluation

4.1 Reliability

Reliability is validated through repeated passes of T1–T3, as well as unit tests for critical modules (M3, M5, M6, M8, M9). Each test ran without failure, confirming robust error handling and internal consistency.

4.2 Usability

The usability of the system was evaluated using a survey as outlined in the VnV Plan (**?**).

5 Comparison to Existing Implementation

The original implementation of ProtGNN (**?**) provided a proof-of-concept model demonstrating prototype-based explanations for graph classification. However, it was not modular, lacked thorough documentation, and did not include automated testing or reproducible training scripts.

The Re-ProtGNN implementation improves upon the original in several key aspects:

- **Modularity and Maintainability:** The system is decomposed into well-defined modules with documented interfaces, following a design guided by Module Guide (MG) and Module Interface Specification (MIS) documentation.
- **Testability:** The re-implementation includes comprehensive unit and system tests (see Sections 4 and 5), with automated test execution through GitHub Actions.
- **Robustness and Reliability:** The training and inference processes were stabilized through added error handling, reproducibility controls (e.g., fixed random seeds), and more consistent logging and checkpoint management.
- **Interpretability Evaluation:** The explanation output was qualitatively similar to the original implementation for MUTAG dataset

(**?**), with subgraphs correctly highlighting chemically meaningful motifs (e.g., aromatic rings and nitro groups).

- **Performance Variability:** While the original code occasionally achieved higher accuracy on MUTAG dataset (**?**), the results were not consistently reproducible. The re-implementation maintains reasonable performance (i.e., within $< 8\%$ decrease in accuracy on average) but prioritizes correctness and explainability over raw accuracy.

Overall, Re-ProtGNN serves as a clean and testable re-engineering of ProtGNN, focusing on transparency, modularity, and reproducibility to support further research and experimentation.

6 Unit Testing

Unit tests were developed for five key modules of Re-ProtGNN. All tests were written using Pytest and were integrated into the CI workflow through GitHub Actions. The testing strategy focuses on functional access routines of each module, aiming for early fault detection.

6.1 Input Format Module (M3)

Unit tests for the input format module are located in tests/test_data_utils.py and verify:

- Correct behavior of load_dataset() under both valid and patched/mock settings
- Robust handling of unsupported dataset names, malformed files, and dataset split configurations
- Construction of DataLoader splits from both random ratios and predefined index masks
- Invocation of dataset wrappers for MUTAG dataset (**?**)
- File I/O functionality (e.g., reading pickled files) for synthetic graph datasets

All tests passed successfully, confirming the correctness of dataset ingestion logic.

6.2 Training Module (M5)

Unit tests for the training module are located in tests/test_train_module.py and verify:

- Loss composition and regularization logic in _compute_total_loss()
- Dataset statistics reporting with _log_dataset_stats()
- Warm-up versus joint training parameter toggling using _set_training_mode()
- Prototype projection behavior using patched get_explanation()
- Accuracy and loss evaluation using _evaluate() on mocked predictions

The top-level train() function is indirectly tested via system test T2 (loss convergence), verifying end-to-end training performance. All unit tests passed.

6.3 Output Visualization Module (M6)

The unit tests in tests/test_output_utils.py validate:

- Logging to file via append_record() using mock_open
- Best checkpoint saving logic in save_best() (both true/false branches)
- Visualization delegation using ExpPlot.draw() with MUTAG routing
- Robustness to unsupported dataset names (raises NotImplementedError)
- Output image generation and disk write confirmation using tmp_path

All unit tests passed, verifying both functional and error-handling behavior for visualization routines.

6.4 Inference Module (M8)

The unit tests in tests/test_output_utils.py and tests/test_inference.py verify:

- Output structure and key consistency of run_inference()
- Accuracy calculation from predicted and true labels

- Logging of inference results
- Batch aggregation correctness for predictions and probabilities

These tests ensure inference correctness under typical and boundary conditions. All unit tests passed.

6.5 Explanation Module (M9)

The tests for this module are in tests/test_explanation_module.py and include:

- Score computation in MCTS nodes (Q, U)
- Prototype similarity scoring between embeddings
- Selection and sorting behavior in node utility functions
- Full MCTS rollout logic with backpropagation
- Explanation interface output type and shape validation

All tests passed, confirming that the explanation module performs as expected.

Summary: The unit tests covered modules M3, M5, M6, M8, and M9, and all tests passed successfully. Modules M1, M2, M4, M7, M10, M11, M12 are either infrastructure-level, configuration-only, or adapted from external libraries, and are verified indirectly through integration tests and execution results.

7 Changes Due to Testing

During the testing phase, several improvements and corrections were made based on the observed behavior of the system.

The most significant change was the removal of the original non-functional requirement specifying that the classification accuracy should exceed 80% on the MUTAG dataset (**?**). While initial experiments showed that the model could occasionally achieve this threshold, repeated testing revealed that the performance was not stable across different random seeds and data splits.

Variability in model convergence, the small size of the MUTAG dataset (**?**), and the sensitivity of prototype-based training contributed to inconsistent accuracy results.

As a result, the accuracy requirement was deemed unreliable as a functional benchmark. It was removed from the Software Requirements Specification (SRS) (**?**) and VnV Plan (**?**), and replaced with a nonfunctional reliability criterion: the model should pass all system and unit tests and show consistent convergence behavior.

This change ensures that the evaluation criteria focus on robustness rather than fixed performance thresholds that may not generalize under limited or variable data conditions.

Other minor updates included clarifying the loss convergence test criteria, refining error-handling in the input format module, and improving test logging in the training and explanation modules.

8 Automated Testing

The Re-ProtGNN project uses GitHub Actions for automated testing and continuous integration. The workflow is triggered on each push or pull request event targeting the main codebase. It performs the following checks:

- Runs all Pytest-based unit tests in the `tests/` directory
- Executes system tests such as:
  - sys_test_input_validation.py – verifies dataset format handling and error messaging
  - sys_test_loss_converge.py – confirms training convergence via decreasing loss trends
  - sys_test_inference.py – evaluates test accuracy and output structure

This automated workflow ensures:

- Unit tests and system tests are run consistently after every commit
- Key training, evaluation, and explanation components remain functional as development progresses

All tests must pass for a pull request to be merged, ensuring stable and reliable builds.

9 Trace to Requirements

The following table shows the traceability between system and unit tests and functional requirements:

Test ID	**R1**	**R2**	**R3**
T1: Missing File Handling Test	X		
T2: Format Consistency Test	X		
T3: Valid Dataset Load Test	X		
T2: Loss Convergence Test		X	
T3: Inference Accuracy Test			X
test-M3-{1–5} (Input Format)	X		
test-M5-{1–5} (Training)		X	
test-M8-{1–4} (Inference)			X
test-M9-{1–6} (Explanation)		X	X

Table 1: Traceability Matrix Between Tests and Functional Requirements

Non-functional requirements were validated through the following means:

- **NFR1 (Reliability)**: Verified via repeatable execution of T1–T3 and all unit tests (M3, M5, M6, M8, M9).
- **NFR2 (Usability)**: Evaluated via usability survey results as outlined in the VnV Plan.

10 Trace to Modules

The following table shows the traceability between tests and software modules:
Note: Modules M10–M12 (PyTorch, PyTorch Geometric, and GUI) are treated as external dependencies and are indirectly verified through the success of system and integration tests involving M3, M5, M6, M8, and M9.

Test ID	M1	M2	M3	M4	M5	M6	M7	M8	M9
T1–T3: Input Validation Tests		X	X						
T2: Loss Convergence Test		X		X	X	X	X		
T3: Inference Accuracy Test		X		X		X	X	X	
test-M3-{1–5} (Input Format)		X	X	X					
test-M5-{1–5} (Training)		X		X	X	X			X
test-M6-{1–6} (Visualization)	X			X		X			
test-M8-{1–4} (Inference)		X		X		X		X	
test-M9-{1–6} (Explanation)		X		X		X			X

Table 2: Traceability Matrix Between Tests and Modules

11 Code Coverage Metrics

Code coverage metrics were collected using `pytest-cov` and `coverage.py` during the automated testing process. The following table summarizes the line-level coverage results for key modules in Re-ProtGNN:

Module	Statements	Missing	Excluded	Coverage
Input Format Module (M3)	151	58	0	62%
Training Module (M5)	103	42	0	59%
Explanation Module (M9)	118	31	0	74%
Inference Module (M8)	26	0	0	100%
Overall (M3, M5, M8, M9)	398	131	0	**67%**

Table 3: Code Coverage for Core Modules

Note on Training Module (M5): The training module has a coverage of 59%. Although its core access routines are unit tested (e.g., loss computation, prototype projection, and evaluation logic), the main training loop involves dynamic control flow, optimizer updates, and state tracking that are harder to isolate. This loop is validated indirectly through the system-level **Loss Convergence Test (T2)** to ensure correct end-to-end training behavior.

Explanation Module (M9) and **Inference Module (M8)** both show high coverage, confirming their correctness under typical and boundary con-

ditions. The **Input Format Module (M3)** also maintains substantial coverage, with untested paths primarily related to legacy dataset support.

The coverage report is automatically generated and archived via GitHub Actions after every CI run.