Thesis proposal

A Unified Machine Learning and Graph-Based Framework for Automated Software Defect Mitigation

Outline for implementing the entire unified framework for automated software defect mitigation:

I. Overall Framework Structure

The implementation will follow a three-phase pipeline:
Defect Prediction (ML-based)
Defect Localization (GAT-based)
Bug Fix Recommendation (RATG-based)

II. Phase 1: Defect Prediction using Ensemble Machine Learning

Data Collection & Preparation:
Obtain datasets (e.g., PROMISE, Defects4J subsets).
Identify software modules (files/classes).
Label modules as defective or clean based on historical bug data.

Feature Extraction:
Calculate static code metrics for each module (e.g., Lines of Code - LOC, Cyclomatic Complexity, Coupling between Objects - CBO, etc.).

Feature Engineering:
Feature Selection: Apply techniques like Maximal Information Coefficient (MIC) and Correlation analysis to remove irrelevant/redundant features and reduce dimensionality.

Addressing Class Imbalance:
Apply the SMOTE-TOMEK hybrid sampling technique to the training dataset to balance the number of defective and clean module samples.

Model Development:
Select base classifiers: Random Forest, Support Vector Machine (SVM), Decision Tree.
Train each base classifier on the refined (selected features + balanced samples) training dataset.
Implement an Ensemble Voting Classifier that combines the predictions of the trained base classifiers.

Model Training: Train the ensemble model using the preprocessed training data.

Model Evaluation:
Evaluate the trained ensemble model on a separate test set.
Use metrics like F1-score, AUC, and Accuracy.
Output: A list of modules/classes or file predicted to be defective.

The evaluation and validation of part of framework will be guided through the following quantitative criteria to ensure that the expected results are meet.

Defect Prediction: F1-score ≥ 85%, AUC ≥ 0.85, Accuracy≥ 85%

Note: for each part of implementation of this phase take the screenshot of output and code run. For example when loading dataset, preprocessing steps (data balancing, feature extraction, feature selection.. ect), training the model, figure of model structure and so on. Also, the out put for evaluation of model like accuracy and lost validation and other evaluation matrices

III. Phase 2: Defect Localization using Graph Attention Network (GAT)

Input Preparation:
Take the source code of the modules flagged as defective in Phase 1.

Code Parsing & Graph Construction:
Parse the source code of each defective module into an Abstract Syntax Tree (AST).

Augment the AST: Add node attributes/features such as:

Static Features:
- Node type (e.g., Loop, Assignment).
- Code metrics (cyclomatic complexity, nesting depth).

Dynamic Features: Defect probability score obtained from Phase 1 for the corresponding module/file.

Convert the augmented AST into a graph representation (nodes represent code elements like statements/functions, edges represent syntactic relationships).

GAT Model Development: The goal is to learn which nodes of AST graph representation are most likely to contain bugs.

Design and implement a Graph Attention Network (GAT) model.
Define the input layer to accept graph-structured data (node features, adjacency information).

Define hidden GAT layers with attention mechanisms to propagate information across the graph.
Define the output layer to produce a "defectiveness" score for each node in the graph.

Model Training:
The model will be trained using datasets with known bug locations (like Defects4J) or potentially be used in a different manner.

Obtain graphs with known buggy nodes.
Train the GAT model on these graphs to learn node-level defectiveness patterns.

Defect Localization:
Feed the augmented graphs (from step 2) into the trained (or appropriately configured) GAT model.
Obtain "defectiveness" scores for each node in the graph.
Rank nodes based on their scores.
Identify the top-N nodes (e.g., statements/lines) as the most likely bug locations within each module.
Output: Specific code locations (nodes) pinpointed as potentially buggy.

The evaluation and validation of part of framework will be guided through the following quantitative criteria to ensure that the expected results are meet.

- Defect Localization: Top-3 localization accuracy ≥ 70%

Note: for each part of implementation of this phase take the screenshot of output and code. For example Code Parsing & Graph Construction, Convert the augmented AST into a graph representation, Model Training, GAT model structure, out put result and so on.

IV. Phase 3: Bug Fix Recommendation using Retrieval-Augmented Template Generation (RATG)

Input Preparation:
Take the buggy code snippets corresponding to the localized nodes/locations from Phase 2.

Retrieval:
Query a database of historical bug fixes (e.g., Defects4J, GitHub commits) using the buggy code snippet. Retrieve the most similar historical bug-fix pairs.

How?

- The robot uses a tool (CodeBERT) to understand code meaning.
- It compares the buggy code to fixes using FAISS (a fast search tool, like Google for code).

Template Generation: Turn Fixes into Reusable Rules (Templates)

For each retrieved historical bug-fix pair:
Perform AST differencing or token differencing between the buggy and fixed code versions.
Abstract the differences into a generalized fix template.

How?
The robot parses code into a tree structure (AST) to spot patterns(Extract AST)

- Use tree-sitter (a parsing tool) to break code into a tree structure.
    1. Parse Code into an AST (Abstract Syntax Tree):
    2. Replace Variables with Placeholders:
        - Original fix: if (b != 0) return a/b;
        - Template: if {denominator} != 0: return {numerator}/{denominator}
    3. Store Templates in a Database: Save templates with metadata (e.g., bug type: DivideByZero).

Fix Application: Match the generalized pattern to the new buggy code and instantiate it (apply it).
Match the buggy code snippet (from Phase 2) against the generated templates using pattern matching techniques (potentially leveraging AST structure).
Apply the matching template(s) to the buggy code to generate candidate repaired code snippets.
Output: Generated candidate code patches/suggestions for the localized bugs.

The evaluation and validation of part of framework will be guided through the following quantitative criteria to ensure that the expected results are meet.

- Bug Fix Recommendation: Valid fix rate (compilation + logic) ≥ 80%

Note: for each part of implementation of this phase take the screenshot of output. For example Template Generation, Fix Application, Templates in a Database, model training, model structure.

V. Integration & Deployment

Pipeline Integration: Connect the output of Phase 1 to the input of Phase 2, and the output of Phase 2 to the input of Phase 3, creating a seamless workflow.

Tool Development: Develop a user interface a web interface to allow users (developers) to input code/project and receive the final output (predicted defective modules, localized bug positions, and suggested fixes).

Evaluation: Evaluate the performance of the entire integrated framework using specified metrics (F1-score, AUC, Accuracy for prediction; Top-3 accuracy for localization; Valid fix rate for repair).

This outline provides the sequential steps and key components involved in implementing the proposed unified framework as described in the thesis proposal.