IT314 - Software Engineering

Prof : Saurabh Tiwari

Group 9 : Pull Panda

(AI-powered Pull Request Reviewer)

Unit-Testing Report

(Regressive unit-testing)

1. Test_accuarcy_checker

This test suite validates the accuracy_checker module. The suite uses mocking to isolate the module from external dependencies (reviewer.py, utils.py) It covers 16 test cases across two main functions: meta_evaluate (7 tests) and heuristic_metrics (9 tests).

`TestMetaEvaluate`

- **`test_meta_evaluate_valid_json`**: Successfully parses clean JSON response with all expected fields (clarity, usefulness, depth, actionability, positivity, explain)
- **`test_meta_evaluate_json_wrapped_in_noise`**: Handles JSON embedded within surrounding text using regex fallback extraction
- **`test_meta_evaluate_invalid_json_inside_braces`**: Gracefully handles malformed JSON that contains braces but invalid syntax, returns error dict
- **`test_meta_evaluate_no_json_anywhere`**: Handles complete absence of JSON structure in response, returns appropriate error message
- **`test_meta_evaluate_llm_exception`**: Catches and handles runtime exceptions from the LLM chain invocation
- **`test_meta_evaluate_truncation_called_properly`**: Verifies that input text is truncated to correct limits (4000 chars for main inputs, 2000 for context inputs) using `safe_truncate`

TestHeuristicMetrics

- **`test_heuristic_basic_counts`**: Correctly counts character length, word count, and bullet points in text
- **`test_heuristic_bug_detection`**: Detects presence of bug-related keywords (bug, error, issue)

- **`test_heuristic_suggest_detection`**: Identifies suggestion-related keywords (suggest, consider)
- **`test_heuristic_no_keywords`**: Confirms keywords are not falsely detected in neutral text
- **`test_heuristic_section_presence_all`**: Recognizes all expected review sections (Summary, Bugs, Errors, Code Quality, Suggestions, Improvements, Tests, Positive, Review)
- **`test_heuristic_section_presence_none`**: Correctly reports absence of all sections in generic text
- **`test_heuristic_empty_text`**: Handles empty string input without errors, returns zero counts
- **`test_heuristic_multiple_bullet_styles`**: Counts different bullet point formats (-, *, •) accurately

2. Test_config

This test suite validates the config.py module. The suite uses pytest.MonkeyPatch to simulate different environment configurations and tests proper loading, validation, error handling, and default behaviors. It covers 8 test cases focusing on environment variable presence, type conversion, validation logic, and error messaging for missing or invalid configurations.

- **`test_config_successful_load_full_env`**: Verifies that all required environment variables (OWNER, REPO, GITHUB_TOKEN, GROQ_API_KEY, PINECONE_API_KEY, PINECONE_INDEX_NAME, PR_NUMBER) are correctly loaded and accessible when properly set
- **`test_config_missing_env_raises_system_exit`**: Ensures that when all required variables are missing or empty, the module raises SystemExit with an error message listing all missing variables
- **`test_config_invalid_pr_number_defaults_to_zero`**: Tests that PR_NUMBER falls back to 0 when set to a non-numeric value that cannot be parsed as an integer
- **`test_config_pr_number_not_set_defaults_to_zero`**: Confirms that when PR_NUMBER environment variable is not provided at all, it defaults to 0
- **`test_config_pr_number_negative_logs_warning`**: Verifies that negative PR_NUMBER values are parsed correctly but trigger a warning message to stdout about invalid PR numbers
- **`test_config_each_variable_missing_individually`**: Parametrized test that checks each required environment variable individually to ensure missing any single variable triggers SystemExit with that specific variable name in the error message (tests OWNER, REPO, GITHUB_TOKEN, GROQ_API_KEY, PINECONE_API_KEY, PINECONE_INDEX_NAME)

3. Test_ingest

This test suite validates the ingest.py module, which handles loading documents from a knowledge base directory, splitting them into chunks, and uploading them to a Pinecone vector store. The suite uses extensive mocking to isolate the module from external dependencies (config, LangChain, Pinecone).It covers 5 test cases focusing on different scenarios in the ingestion workflow.

- **`test_ingest_no_documents`**: Verifies that when the DirectoryLoader finds no documents in the knowledge base directory, the function prints a "No documents found" message and exits gracefully without attempting further processing
- **`test_ingest_full_flow`**: Tests the complete happy path where documents are loaded (1 document), split into chunks (2 chunks), and uploaded to an existing Pinecone index, confirming all steps execute in sequence and the correct output messages are printed
- **`test_ingest_create_index`**: Ensures that when the specified Pinecone index doesn't exist in the list of available indexes, the function creates a new index before attempting to upload document chunks
- **`test_main_directory_missing`**: Tests the `__main__` block behavior when the knowledge base directory doesn't exist on the filesystem, verifying that an appropriate error message is printed
- **`test_main_directory_exists`**: Validates the `__main__` block behavior when the knowledge base directory exists, ensuring that the `ingest_data()` function is called to begin the ingestion process

4. Test_interative_prompt_selector

This comprehensive test suite validates the iterative_prompt_selector.py module, which implements an adaptive machine learning system for selecting optimal code review prompts based on PR characteristics The suite uses extensive mocking to isolate all external dependencies and covers 13 test classes with 90+ test cases, testing initialization, feature extraction, model training, review generation, state persistence, and the complete PR processing workflow including edge cases and boundary conditions.

`TestIterativePromptSelectorInit`

- **`test_initialization_creates_all_components`**: Verifies that initialization creates all required components including prompts dictionary, model (SGDRegressor), scaler (StandardScaler), and initializes training state variables
- **`test_initialization_creates_empty_histories`**: Confirms that feature_history, prompt_history, and score_history lists are initialized as empty
- **`test_retriever_initialization`**: Ensures the RAG retriever is properly initialized and callable during setup

`TestExtractPRFeatures`

- **`test_basic_diff_features`**: Tests extraction of basic metrics (line count, file count, additions, deletions) from a simple git diff
- **`test_empty_diff`**: Handles empty string diff, returning zero counts for files/additions/deletions
- **`test_multiple_files_detection`**: Correctly counts number of files when diff contains multiple "diff --git" headers
- **`test_python_file_detection`**: Identifies Python files (.py extension) and sets is_python flag to 1
- **`test_javascript_file_detection`**: Identifies JavaScript files (.js extension) and sets is_js flag to 1

- **`test_typescript_file_detection`**: Identifies TypeScript files (.ts extension) as JavaScript files
- **`test_java_file_detection`**: Identifies Java files (.java extension) and sets is_java flag to 1
- **`test_function_detection`**: Detects presence of function definitions (def, function keywords) in diff
- **`test_import_detection`**: Detects import statements (import, from...import) in code
- **`test_comment_detection`**: Identifies various comment styles (#, //, /* */) in the diff
- **`test_test_file_detection`**: Detects test files (test_ prefix, _test suffix, /tests/ directory)
- **`test_documentation_detection`**: Identifies documentation files (README, .md, .rst extensions)
- **`test_config_file_detection`**: Detects configuration files (.json, .yaml, .toml, .ini, .cfg extensions)
- **`test_net_changes_calculation`**: Calculates net changes as additions minus deletions
- **`test_large_diff_handling`**: Properly handles large diffs with thousands of lines without performance issues

`TestFeaturesToVector`

- **`test_complete_features_conversion`**: Converts complete feature dictionary to 14-element numpy array in correct order
- **`test_missing_features_default_to_zero`**: Missing features in dictionary default to zero in the output vector
- **`test_empty_features_dict`**: Empty dictionary produces all-zero vector of correct length
- **`test_extra_features_ignored`**: Extra keys in feature dictionary that aren't in expected feature set are safely ignored

`TestSelectBestPrompt`

- **`test_first_sample_selection`**: For the first sample, always selects the first prompt (exploration phase)
- **`test_second_sample_selection`**: For the second sample, selects the second prompt (continued exploration)
- **`test_selection_with_fitted_scaler`**: Uses fitted scaler and trained model to predict best prompt after sufficient training data
- **`test_exploration_policy_early_samples`**: Implements epsilon-greedy exploration during early training phase
- **`test_prediction_failure_handling`**: Falls back to random selection when model prediction fails
- **`test_scaler_transform_failure`**: Handles scaler transformation errors gracefully with fallback selection

`TestUpdateModel`

- **`test_first_update`**: First model update adds data to histories and increments sample count
- **`test_scaler_fitting_after_two_samples`**: StandardScaler is fitted after collecting two or more samples
- **`test_multiple_updates`**: Handles 10 sequential updates, maintaining correct history lengths and sample count
- **`test_model_reinitialization_on_failure`**: Reinitializes SGDRegressor model when partial_fit fails
- **`test_scaler_refit_on_transform_failure`**: Refits scaler when transformation fails during model update

`TestGenerateReview`

- **`test_successful_review_generation`**: Successfully generates review using selected prompt, LLM chain, static analysis, and RAG context
- **`test_static_analysis_failure_handling`**: Catches static analysis errors and returns error message in static_output
- **`test_rag_retrieval_failure_handling`**: Handles RAG retrieval failures and returns error message in context field

- **`test_llm_invocation_failure_handling`**: Catches LLM chain invocation errors and returns error message as review text
- **`test_truncation_applied`**: Verifies that safe_truncate is called multiple times to limit input sizes for diff, static analysis, and context

`TestEvaluateReview`

- **`test_successful_evaluation`**: Combines heuristic metrics and meta-evaluation to compute review quality score between 0-10
- **`test_evaluation_with_meta_error`**: When meta-evaluation returns error, defaults to score of 5.0
- **`test_evaluation_score_calculation`**: Tests weighted combination of heuristic features and meta-evaluation scores
- **`test_evaluation_with_short_review`**: Handles evaluation of very short reviews with minimal features
- **`test_evaluation_with_long_review`**: Handles evaluation of very long reviews (1500+ words) without errors

`TestSaveState`

- **`test_save_state_success`**: Successfully serializes and saves training state (histories, sample count) to JSON file
- **`test_save_state_with_fitted_model`**: Includes model coefficients, intercept, scaler parameters in saved state when model is fitted
- **`test_save_state_io_error`**: Returns False when file I/O error occurs during save
- **`test_save_state_with_empty_history`**: Successfully saves even when no training data exists yet

`TestLoadState`

- **`test_load_state_file_not_found`**: Returns False when attempting to load non-existent state file
- **`test_load_state_success`**: Successfully loads training histories, sample count, and scaler status from JSON file

- **`test_load_state_with_scaler`**: Restores fitted scaler parameters (mean and scale) from saved state
- **`test_load_state_with_model_weights`**: Restores model coefficients and intercept from saved state
- **`test_load_state_combines_with_existing`**: Merges loaded state with existing in-memory data rather than overwriting
- **`test_load_state_avoids_duplicates`**: Prevents adding duplicate samples when loaded data matches existing data
- **`test_load_state_invalid_json`**: Returns False when JSON file is malformed or corrupted
- **`test_load_state_corrupted_scaler_data`**: Handles corrupted scaler data gracefully and continues loading other valid data

`TestProcessPR`

- **`test_process_pr_success`**: Executes complete PR processing workflow: fetch diff, extract features, select prompt, generate review, evaluate, save results
- **`test_process_pr_posts_to_github`**: When post_to_github=True, posts generated review as comment to GitHub PR
- **`test_process_pr_github_post_failure`**: Continues successfully even when GitHub comment posting fails
- **`test_process_pr_fetch_diff_failure`**: Returns error result with score 0 when PR diff cannot be fetched
- **`test_process_pr_saves_state_periodically`**: Calls save_state after processing when sample count reaches threshold
- **`test_process_pr_with_custom_params`**: Accepts custom owner, repo, and token parameters for GitHub API calls

`TestSaveResults`

- **`test_save_results_success`**: Saves both JSON (structured data) and Markdown (human-readable) result files
- **`test_save_results_creates_proper_filenames`**: Creates filenames with PR number and appropriate extensions (.json, .md)

- **`test_save_results_includes_all_data`**: JSON output includes pr_number, prompt, score, heuristics, meta-evaluation, static analysis, and RAG context

`TestGetStats`

- **`test_get_stats_empty_history`**: Returns zero values for all statistics when no training has occurred
- **`test_get_stats_with_data`**: Computes correct average score and unique prompt count from training history
- **`test_get_stats_prompt_distribution`**: Calculates distribution showing how many times each prompt was used
- **`test_get_stats_scaler_status`**: Includes scaler fitted status in returned statistics dictionary

`TestRunIterativeSelector`

- **`test_run_iterative_selector_single_pr`**: Processes single PR and returns results list and selector instance
- **`test_run_iterative_selector_multiple_prs`**: Processes multiple PRs sequentially, returning all results
- **`test_run_iterative_selector_loads_previous_state`**: When load_previous=True, loads saved state before processing
- **`test_run_iterative_selector_periodic_saves`**: Saves state at specified frequency during multi-PR processing
- **`test_run_iterative_selector_handles_pr_failure`**: Continues processing remaining PRs when one fails with exception
- **`test_run_iterative_selector_final_save`**: Performs final state save after completing all PR processing

`TestEdgeCasesAndBoundaries`

- **`test_very_large_diff`**: Handles diffs with 100,000+ lines without crashes or memory issues

- **`test_diff_with_special_characters`**: Properly processes diffs containing Unicode characters and emojis
- **`test_zero_features_vector`**: Handles all-zero feature vector and still selects valid prompt
- **`test_negative_score_update`**: Accepts and stores negative scores in training history
- **`test_very_high_score_update`**: Accepts and stores scores above normal range (>10) in training history
- **`test_nan_in_features`**: Handles NaN values in feature vector without crashing
- **`test_empty_prompt_name`**: Raises ValueError when attempting to update model with empty prompt name
- **`test_concurrent_updates_simulation`**: Handles 100 rapid sequential updates maintaining data integrity

5. Test_rag_core

This test suite validates the rag_core.py module, which provides a caching layer for Retrieval-Augmented Generation (RAG) components including embeddings, vector stores, and retrievers for Pinecone. The suite uses lightweight fake modules to replace heavy dependencies (HuggingFace, Pinecone, LangChain) and tests caching behavior, initialization logic, and error handling. It covers 7 test cases focusing on cache behavior, first-time initialization, and configuration validation.

Embeddings Management Tests

- **`test_get_embeddings_first_call`**: Verifies that first call to `_get_embeddings()` creates a new HuggingFaceEmbeddings instance with the correct model name and stores it in the cache
- **`test_get_embeddings_cached`**: Ensures that subsequent calls to `_get_embeddings()` return the same cached instance rather than creating new objects

Vector Store Management Tests

- **`test_get_vector_store_first_call`**: Tests that first call to `_get_vector_store()` initializes a PineconeVectorStore from existing index with correct index name and stores it in the cache
- **`test_get_vector_store_cached`**: Confirms that subsequent calls to `_get_vector_store()` return the same cached vector store instance
- **`test_get_vector_store_missing_api_key`**: Validates that attempting to create a vector store without a valid PINECONE_API_KEY raises a ValueError with appropriate error message

Retriever Management Tests

- **`test_get_retriever_first_call`**: Verifies that first call to `get_retriever()` creates a retriever with specified k_value parameter and caches the result
- **`test_get_retriever_cached`**: Ensures that subsequent calls to `get_retriever()` return the same cached retriever instance without reinitializing

6. Test_reviewer

This test suite validates the reviewer.py module.The suite uses fake modules to mock heavy dependencies (LangChain, Groq) and monkeypatching to simulate various HTTP responses and error conditions. It covers 11 test cases across three main functions: fetch_pr_diff (6 tests), post_review_comment (3 tests), and save_text_to_file (2 tests), testing both successful operations and comprehensive error handling.

`fetch_pr_diff`

- **`test_fetch_pr_diff_success`**: Verifies successful retrieval of PR diff text when GitHub API returns 200 status code
- **`test_fetch_pr_diff_404`**: Handles 404 Not Found response when PR doesn't exist, returns empty string and prints "Could not find PR" message
- **`test_fetch_pr_diff_401`**: Handles 401 Unauthorized response due to invalid GitHub token, returns empty string and prints "Invalid GitHub Token" message
- **`test_fetch_pr_diff_other_http_error`**: Handles other HTTP errors (e.g., 500 server error) that are neither 401 nor 404, returns empty string and prints "GitHub API Error" message
- **`test_fetch_pr_diff_network_error`**: Catches network-related exceptions (RequestException) when API call fails, returns empty string and prints "Unexpected network error" message

`post_review_comment`

- **`test_post_review_comment_success`**: Successfully posts comment to GitHub when API returns 201 status, returning the JSON response data
- **`test_post_review_comment_api_error`**: Raises RuntimeError when GitHub API returns error status (400+) indicating request failure
- **`test_post_review_comment_network`**: Raises RuntimeError when network exception occurs during the POST request

`save_text_to_file`

- **`test_save_text_to_file_success`**: Successfully writes text content to file with UTF-8 encoding using temporary directory
- **`test_save_text_to_file_oserror`**: Handles OSError during file write operation, prints error message instead of crashing

7. Test_static_analysis

This test suite validates the static_analysis.py module, which extracts changed files from git diffs and runs language-specific static analysis tools (like Pylint, ESLint, Cppcheck) on them. It covers 12 test cases across two main functions: get_changed_files_and_languages (3 tests) and run_static_analysis (9 tests), ensuring robust handling of different file types, analyzer outputs, and failure scenarios.

`get_changed_files_and_languages`

- **`test_get_changed_files_and_languages_normal`**: Correctly extracts multiple files from diff and groups them by language (Python, JavaScript, C++) based on file extensions
- **`test_get_changed_files_and_languages_no_match`**: Returns empty dictionary when diff contains only files with no recognizable programming language extensions (e.g., README.md)
- **`test_get_changed_files_and_languages_multiple_same_lang`**: Groups multiple files of the same language together, handling case-insensitive extensions (.py and .PY)

`run_static_analysis`

- **`test_run_static_analysis_no_changed_files`**: Returns message "No recognizable programming language files" when diff contains only non-code files
- **`test_run_static_analysis_lang_with_no_analyzer`**: Returns message "No analyzer configured" when a detected language has no configured analyzer tool
- **`test_run_static_analysis_success_stdout`**: Successfully captures and returns stdout output from analyzer tool, including analyzer name (e.g., "Pylint")
- **`test_run_static_analysis_success_stderr`**: Captures and returns stderr output when analyzer writes results to standard error instead of stdout

- **`test_run_static_analysis_success_no_output`**: Returns "No issues found" message when analyzer completes successfully but produces no output
- **`test_run_static_analysis_filenotfound`**: Handles FileNotFoundError when analyzer binary is not installed or not in PATH, returns "Command not found" message
- **`test_run_static_analysis_timeout`**: Catches TimeoutExpired exception when analyzer exceeds execution time limit, returns "Execution timed out" message
- **`test_run_static_analysis_generic_exception`**: Handles unexpected RuntimeError or other exceptions during analyzer execution, returns "Error running analyzer" message

8. Test_utils

This test suite validates the `utils.py` module's `safe_truncate` function, which intelligently truncates long text while preferring to break at newline boundaries for cleaner output.The suite covers 7 test cases testing various truncation scenarios including normal truncation, edge cases with newlines, boundary conditions, and special cases like empty strings and extreme length limits.

`safe_truncate`

- **`test_safe_truncate_no_truncation`**: Returns original text unchanged when it's shorter than the specified maximum length
- **`test_safe_truncate_truncate_with_newline`**: When text exceeds max length and contains newlines, truncates at the last newline before the limit and appends "... (Output truncated)" suffix
- **`test_safe_truncate_truncate_without_newline`**: When text exceeds max length but contains no newlines, truncates at the character limit and appends "... (Output truncated)" suffix
- **`test_safe_truncate_exact_newline_on_limit`**: When a newline character sits exactly at the truncation boundary, truncates cleanly at that newline with truncation suffix
- **`test_safe_truncate_max_len_zero`**: Handles edge case where max_len is 0, returns empty prefix with truncation suffix
- **`test_safe_truncate_empty_string`**: Returns empty string unchanged when input is empty (no truncation needed)
- **`test_safe_truncate_one_char_limit`**: Correctly truncates to single character when max_len is 1, preserving first character with truncation suffix

Final summary

Module	Coverage%	Test Cases	Status
accuracy_checker	100	16	complete
config	100	8	complete
ingest	100	5	complete
iterative_prompt_selector	93	90+	near complete
rag_core	100	7	complete
reviewer	100	11	complete
static_analysis	100	12	complete
utils	100	7	complete
Total	**100**	**149+**	**Excellent**