Samsung Semiconductor India Research

2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Beyond Heuristics: AI/ML Driven Verification for Design Sign-off

Sougata Bhattacharjee, Gulshan Kumar Sharma, Abhishek Raj,
Avinash Bollu, and Akshaya Kumar Jain
Samsung Semiconductor India Research (SSIR)

Problem Statement and Motivation

Problem:

- Regression management and Coverage Closure
- Hard to hit Bins or cover bins
- Multiple merging of databases
- Bug Hunting
- More Compute resource
- Redundant or less-used Test Scenarios
- Meeting time to market

Solutions:

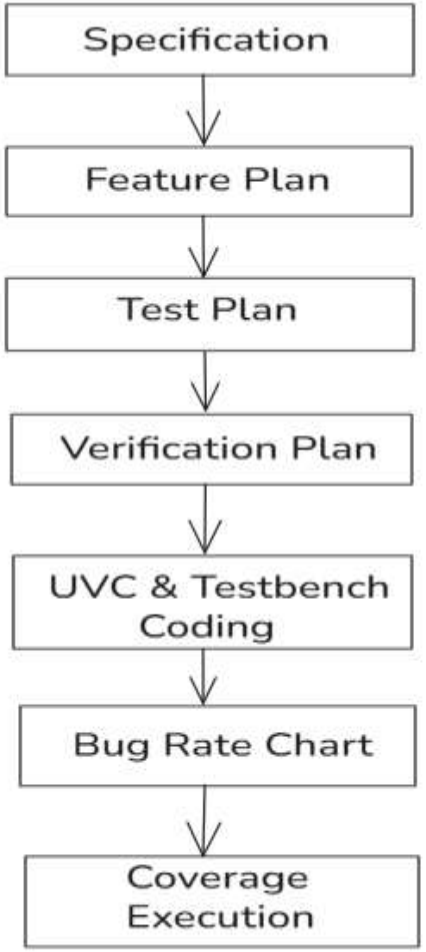
- AI/ML Driven Methodology
- Adaptive Learning approach

Agenda

- Traditional Verification Flow
- Ranking approach for Coverage Closure
- AI/ML based Methodology
- Different approach in AI/ML – Tool based and Custom Pipeline
- Custom Pipeline Flow
- Results
- Conclusion

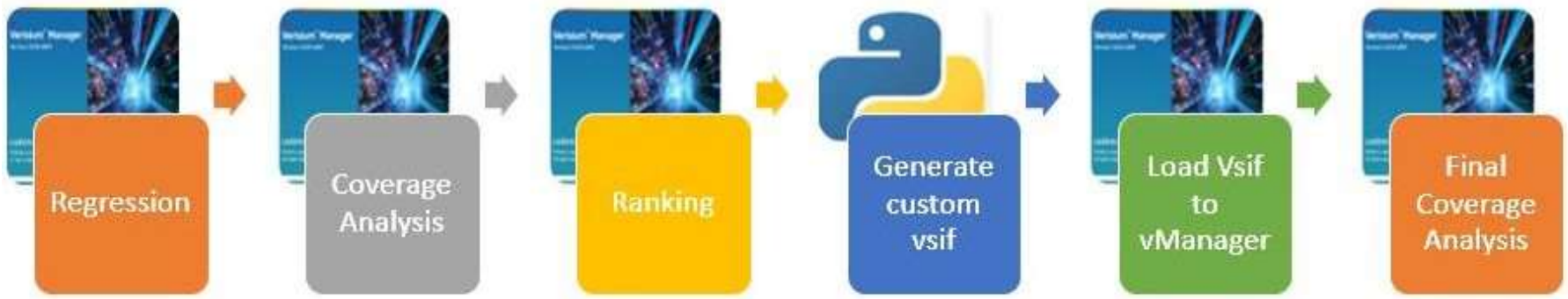
Basic Traditional Verification Flow

- Microarchitecture Specification
- Feature Extraction related to different features of IP
- Test Scenarios to be coded
- Test bench Infrastructure description and related to UVCs
- UVC coding
- Test Scenario execution
- Data and Design Integrity using Scoreboard
- Root Cause Analysis and Bug Fixes
- Regression Management and Coverage Holes
- Coverage maximization and Bug Hunting



Ranking Approach

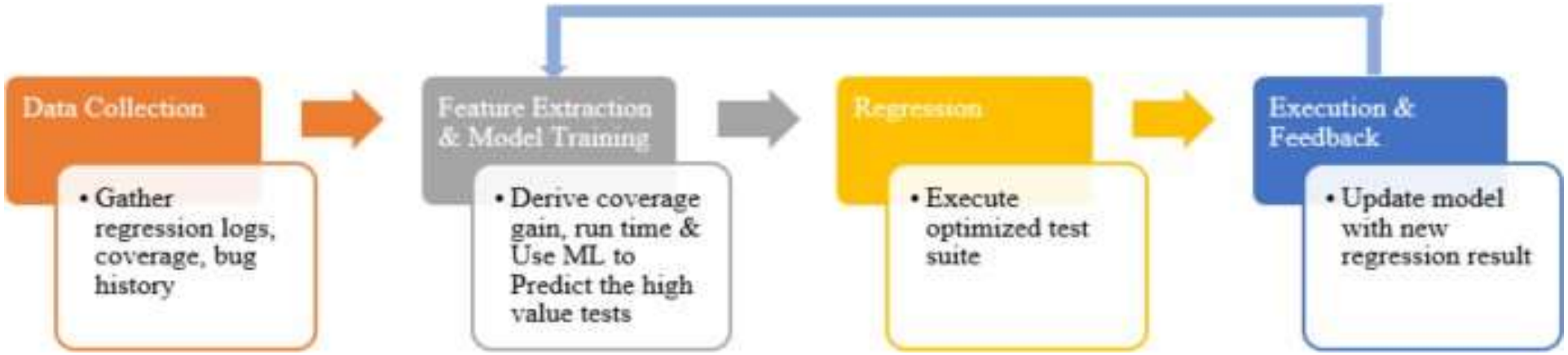
- Creates Testlist in ascending order based on DUT Coverage
- High-value tests are collected and given as input to scripts to generate a customized test list with a different seed number for each test
- Helps improve regression efficiency and coverage over the traditional approach



Ranking and Resequencing Flow

Custom Pipeline Methodology

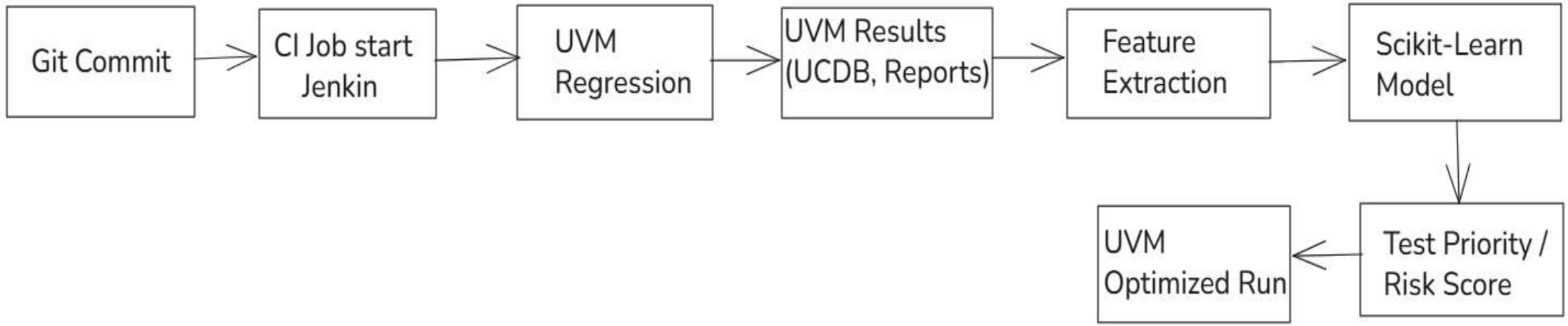
- Regression data collection – Test, Seed, Outcome, Runtime, Coverage delta
- Model training using ML and randomizing parameter
- Generation of optimized test list
- Iteration with feedback and optimized regression



Proposed Methodology Flow

Feature Extraction

- Feature Extraction is the process of converting raw UVM data, logs and reports into numerical signals that represent test behaviour and verification value
- A simplified flowchart for feature extraction is shown below:



Execution steps of Feature Extraction

[1] CI Jobstart (Jenkin):

- Jenkins is an open source automation server used to implement a CI/CD pipeline by automating repetitive tasks, testing, and deploying code
- In the context of UVM, it orchestrates the execution order and automates regression, distributed testing and provides feedback about the reports, which will later be used in an optimized run
- A simple snippet of Jenkins Groovy extension is shown below, which is provided as an input to feature verification after the UVM regression run

```
pipeline {
    agent any

    stages {
        stage('Run UVM Regression') {
            steps {
                sh 'make regress'
            }
        }

        stage('Feature Engineering + ML') {
            steps {
                sh 'python3 ml/train_predict.py'
            }
        }
    }
}
```

Execution steps of Feature Extraction Cont'd

[2] Feature Engineering:

- Before feature engineering, the raw regression history looks like this:

```
run_id, test_name,   pass, run_time, coverage, timestamp
57, axi_random_test, 0,        235,       86.1,      2026-01-03
……..
```

- The process of feature engineering aggregates each row and makes one feature vector per test, and calculates the features mentioned below:

[a] Failure Likelihood = Historical probability of failures

```
fail_rate = 1 - df["pass"].mean()
```

[b] Runtime = Is it a long-running test or a test with a shorter duration

```
runtime = df["run_time"].mean()
```

Execution steps of Feature Extraction Cont'd

[c] Execution Recency = How frequently the test has run. Did it run after the RTL change?

Depending on that, the test will be prioritized or will be removed

```
last_run = df["timestamp"].max()
```

[d] Seed Diversity Index = Entropy-based measure of outcome for random seeds

It categorizes the test into High Diversity, Low Diversity, and Diversity with Failures, and calculates it based on Shannon Entropy.

$H = - \Sigma\ p(x) * \log2(p(x))$

```
outcomes = df["pass"].value_counts(normalize=True)

entropy = -np.sum(outcomes * np.log2(outcomes))
```

Scikit-learn Model

- The input data from feature extraction is categorised into rows and columns, which are the inputs to the model.

Row → test_name

Columns → Failure likelihood, runtime, execution recency, and Seed Diversity Index

- Feature Matrix is prepared depending on the data set of the tests.

X → The parameters mentioned above

Y → Is the scenario passed or failed?

In short, P(fail | features) → will provide the test or risk score

- The learning of the model happens through three different models:

  [a] Logistic regression

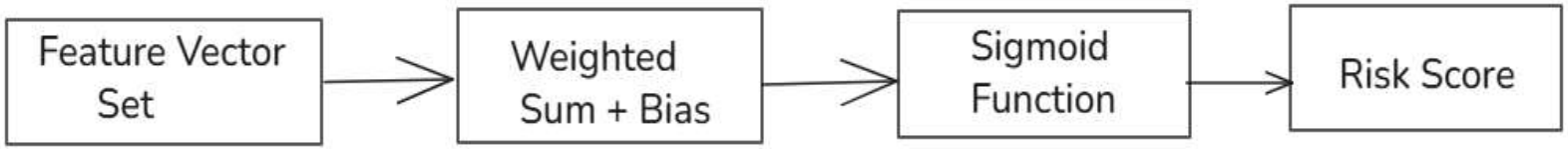
  [b] Random Forest

  [c] Gradient Boosting

Scikit-learn Model Cont'd

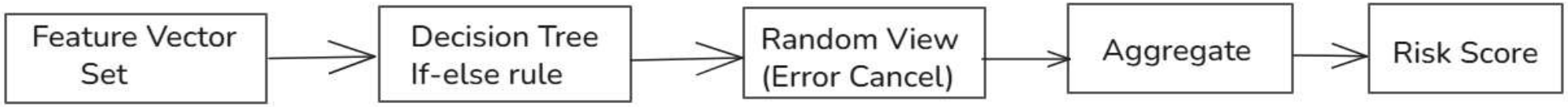
Logistic Regression:

- Best used for coverage goals with a hit/miss ratio.
- Calculate based on Weighted Risk estimation



Random Forest:

- Best used for predicting continuous coverage metrics
- Handles non-linear prediction using aggregate rules.



Gradient Boosting:

- Sequentially add small models that focus on correcting the mistake of the previous model
- Boosting happens through assigning the difficult test more weight, easy scenarios are executed less & repeat

Optimized UVM Run

This process comprises the following steps:

- Regression results are appended to the database after every run.
- Coverage, failures, and risk scores are extracted.
- The model trains itself after every run.

In short, it uses ML-derived risk scores to intelligently select tests to be part of the regression analysis.

- SIM AI tool has also been explored to automate log extraction, simulation data, and UCDB logs, but all the ML processing and regression execution have been executed with a custom pipeline for better control, transparency, and to prevent overfitting.

Automated flow Steps

- Git Commit RTL and TB
- Jenkins Installation and related plugin
- CI control script (Groovy) → Consists of a sequence of steps related to Feature extraction, ML prediction, etc
- Regression script
- Coverage normalization
- Python and libraries installation like numpy, pandas and scipy.
- Install ML libraries like Scikit learn and joblib
- Model training script importing Random Forest Classifier
- Test selection logic depending on the risk score
- Optimized UVM Regression

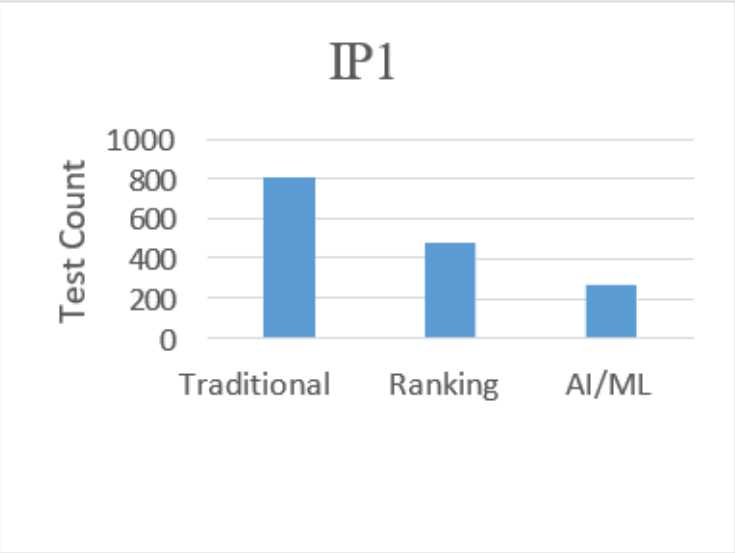
Application of Proposed Methodology

- Coverage optimisation and faster Coverage Closure.
- Reduced redundant simulations.
- Clean Regression
- Higher bug hunting probability
- Continuous Verification
- Better resource planning
- No Manual Testlist Creation

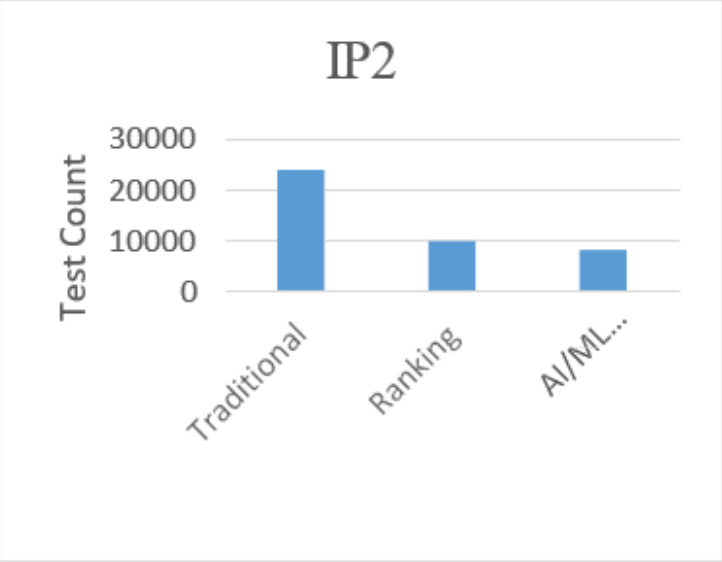
Results

[1] Test Count Optimisation:

- The figures below comprise the Test Count in 2 Multimedia IPs.
- IP1 has around 814 tests (including seed count), and IP2 has 24000 tests (including seed count)
- With the help of AI/ML methodology, the test count has been reduced to ~65% to achieve coverage.



Test count reduction in IP1

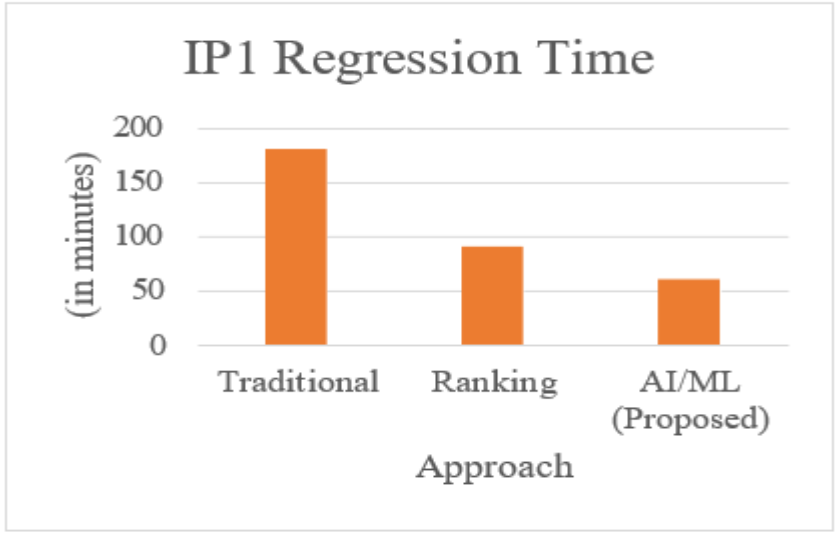


Test count reduction in IP2

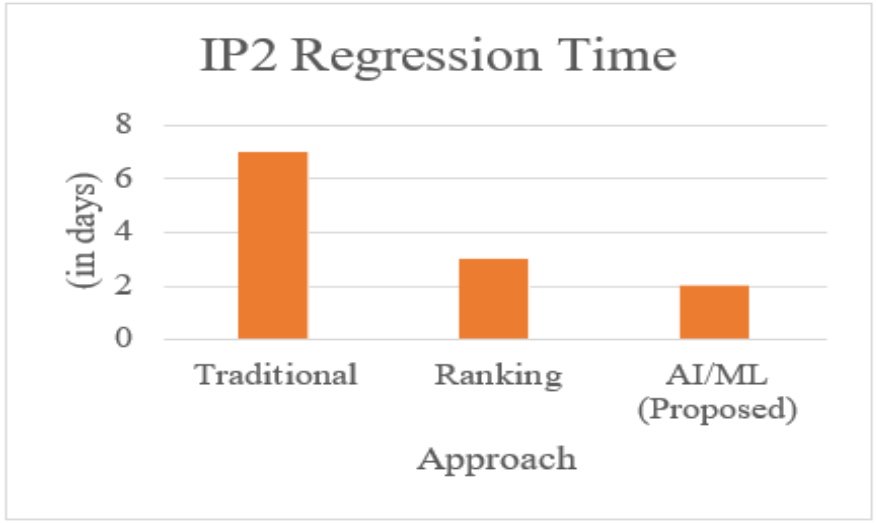
Results Cont'd

[2] Regression Efficiency:

- The figures below show the regression efficiency in terms of reduced regression time.
- Regression runtime reduced by ~66% for IP1 and ~72% for IP2 and ~31% in both IP1 and IP2 through ranking.



Regression Efficiency in IP1



Regression Efficiency in IP2

Results Cont'd

[3] Coverage Closure:

- Achieved 92.7% coverage for IP1 by regaining 286,354 bins out of 308,999 target bins.
- Achieved coverage of 91.6% with 67% lesser seeds and 71% lesser time than random execution, and 18.5% less seeds and 33% less time than Ranking for IP2.

[4] Hard to cover bins:

- Learns and creates vsif to target uncovered bins.
- Covered bins increase with each iteration.
- An iterative process leverages coverage feedback to increase hit efficiency.


```
{
    "num_runs_init_iter": 814,
    "num_covered_bins_init_iter": 1348672,
    "newly_hit_original_holes_count": 1416,
    "newly_hit_original_holes": [
        "Instances/top/dut/GIC400_WRAPPER/GIC400/u_gic_distributor/g_spis[2]/u_spi/42/1/4",
        "Instances/top/dut/GIC400_WRAPPER/GIC400/u_gic_distributor/g_spis[2]/u_spi/43/1/8",
```


Learning based approach to cover hard-to-hit bins


```
{
    "Original dataset stats": {
        "Number of bins": 448509,
        "Number of runs": 814,
        "# bins not covered": 139510,
        "# bins covered": 308999,
        "# bins always hit": 44093,
        "# rare bins (<0.05)": 53742,
        "Total CPU time [hrs]": 52.944
    },
    "Result dataset stats": {
        "Number of bins": 448509,
        "Number of runs": 244,
        "# bins not covered": 160894,
        "# bins covered": 287615,
        "# bins always hit": 186416,
        "# rare bins (<0.05)": 33709,
        "Total CPU time [hrs]": 11.323
    },
    "Comparison": {
        "Target coverage": 308999,
        "New coverage": 287615,
        "# new hits": 1261,
        "# regained bins": 286354,
        "# NOT regained bins": 22645,
        "Num rare bins in original dataset": 53742,
        "Num regained rare bins": 33240
    }
}
```

Results Cont’d

[5] <u>Resource Utilization:</u>

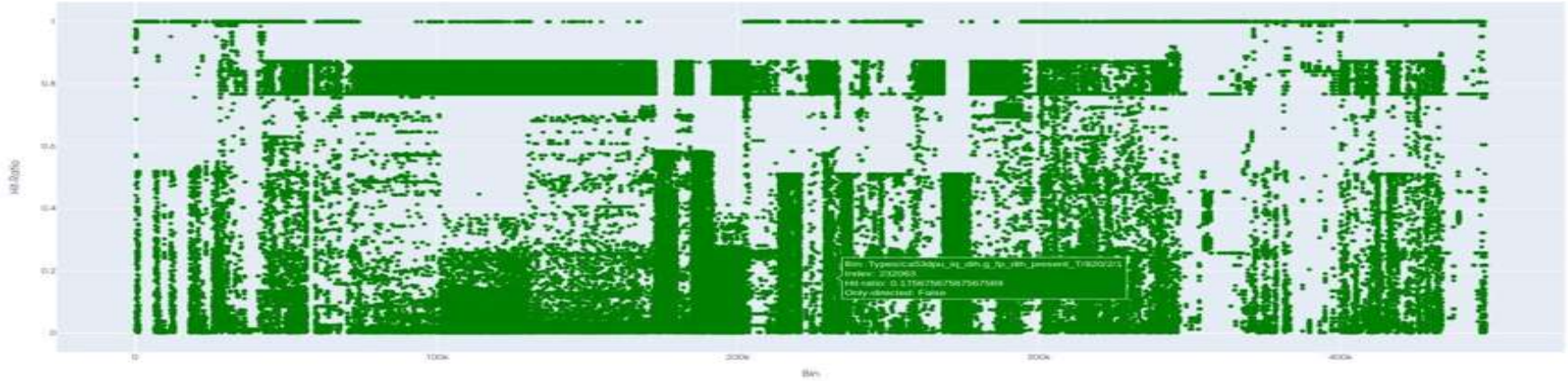
- As the redundant tests are removed from the optimized test suite, the requirement of compute resource also decreases.

[6] <u>Generalization across Architecture:</u>

- Domain independence is validated by reusing the same framework across different coverage models with ~60% reduction in regression efficiency.

[7] <u>Analysis Efficiency with Analytics:</u>

- The below figure represents a 2D plot view, where selecting a specific point in the plot, the corresponding hit ratio along with the test and its seed can be visualized.



Conclusion

Aspect	Traditional	Ranking	AI/ML (Proposed)
Regression Efficiency	Complete regression running	Relies on manual test selection, also removes redundant runs	ML model prioritize high value tests, reducing redundant runs
Coverage Closure	Takes a long time	Faster but manual tuning	Faster & Automated
Stressed Testing	Manual	Manual	Automated
Resource Utilization	Poor	Moderate	Optimum
Adaptability	Low	Moderate	High, Self-learning

THANK YOU