A Learning-to-Rank Based Fault Localization Approach

Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske

18/02/2025

Index

Overview

- Motivation:
  - Debugging is expensive in time and effort.
  - Existing fault localization techniques generate long candidate lists.

- Idea pitched:
  - **Savant** - a fault localization approach that uses a learning-to-rank strategy.

- Core Components:
  - Consists of four steps:
    1. **Method Clustering & Test Case Selection**
    2. **Invariant Mining**
    3. **Feature Extraction**
    4. **Method Ranking**

- Results
  - Evaluated on 357 real bugs from 5 Defects4J projects.
  - Improves localization accuracy significantly (e.g., 57.73% better at top-1 ranking).

Index

No	Content
1.	Overview
2.	**Preliminaries**
3.	Example
4.	Proposed Approach
5.	Experiments
6.	Research Questions and Findings
7.	Conclusion and Future Work
8.	Q/A

Preliminaries

1. Spectrum-Based Fault Localization (SBFL)
2. Mining Likely Invariants (using Daikon)
3. Learning-to-Rank Techniques

Preliminaries

1. <u>Spectrum-Based Fault Localization (SBFL)</u>

- Goal: Rank program elements by likelihood of being buggy.
- Analogy:
  - Like finding out which ingredient made people sick at a dinner party: If everyone who ate the cake got sick, but those who skipped it were fine → the cake is suspicious!
- Method:
  - Use execution data from passing and failing test cases.
  - Compute statistical metrics (suspiciousness scores).
- Key Idea:
  - Elements executed frequently by failing tests—but rarely by passing tests—are more suspicious.
- Pros & Cons:
  - Fast, automatic, no deep code analysis.
  - Can’t always pinpoint the exact bug (just highlights suspicious areas).
- Why It Matters for Savant:
  - Savant uses SBFL’s “suspiciousness scores” as one clue but combines it with invariant violations for better accuracy.

Preliminaries

1. <u>Mining Likely Invariants (using Daikon)</u>

- Purpose:
  - Capture the expected behavior of a program—think of it as understanding what "normal" looks like for the code.
- Tool: Daikon – a widely-used system that automatically detects these "normal behavior" rules (invariants).
- How It Works:
  - It monitors the values of variables at specific points in the program (like at the beginning or end of a function).
  - It compares these observed values to a huge list (over 300) of pre-defined "rules" or templates to see which ones hold true.
- Examples of Invariants:
  - LowerBound: Ensures a variable is always at least a certain value (e.g., $x \geq c$).
  - LinearBinary: Checks for a consistent linear relationship between two variables (e.g., $ax + by + c = 0$).
  - NonZero: Verifies that a variable is never zero or null.

Preliminaries

1. Learning-to-Rank Techniques

- What It Is:
  - A set of supervised machine learning methods for ranking items.
- Phases:
  - Learning Phase: Extract features from training data (with known buggy methods) to build a ranking model.
  - Deployment Phase: Use the model to rank program elements for new bugs.
- Our Approach:
  - Utilizes rankSVM, a pairwise ranking algorithm.
- Why It Matters:
  - Combines diverse features (invariant differences and SBFL scores) into a single ranked list.

Index

No	Content
1.	Overview
2.	Preliminaries
3.	**Example**
4.	Proposed Approach
5.	Experiments
6.	Research Questions and Findings
7.	Conclusion and Future Work
8.	Q/A

Example

- Bug Context:
  - Bug 383 from the Closure Compiler bug database.
  - High priority bug affecting Internet Explorer 9 and jQuery.getScript.
- Bug Description:
  - Incorrect translation of string constants (e.g., "\0", "\x00", "\u0000").
  - Expected: A string literal with “\0" (or similar).
  - Observed: A string literal with three null characters.
- Developer Patch:
  - Bug resides in the strEscape method of com.google.javascript.jscomp.CodeGenerator.
- How Savant Helps:
  - Savant focuses on the differences in program behavior (invariants) between passing and failing tests.
  - For Bug 383, it quickly narrows down which methods act differently under failing conditions and then ranks them using a learning-to-rank model.
  - This approach dramatically cuts down the list of suspects from thousands of methods to just a handful, pushing the faulty strEscape method to the top, making it far easier for developers to spot and fix and not look at 6,646 other failing methods.

Bug 383 (Priority: high)

Summary: \0 \x00 and \u0000 are translated to null character

Description:
What steps will reproduce the problem?
1. write script with string constant “\0” or “\x00” or “\u0000”

What is the expected output? What do you see instead?
I expected a string literal with “\0” (or something like that) and instead get a string literal with three null character values.

Please provide any additional information below.
This is causing an issue with IE9 and jQuery.getScript. It causes IE9 to interpret the null character as the end of the file instead of a null character.

```
@@ -963,6 +963,7 @@ class CodeGenerator {
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        switch (c) {
+         case '\{}0': sb.append("\{}\{}0"); break;
          case '\{}n': sb.append("\{}\{}n"); break;
          case '\{}r': sb.append("\{}\{}r"); break;
```

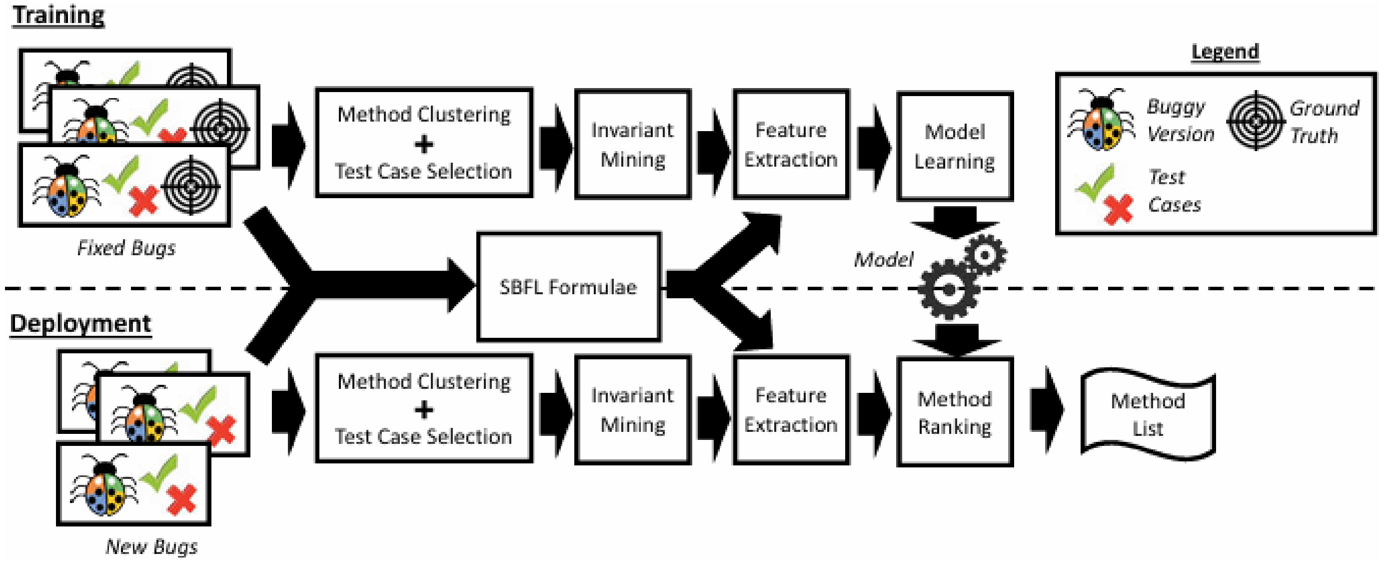
Figure 1: Bug Report (top) and developer patch (bottom) for bug 383 of the Closure Compiler

Index

No	Content
1.	Overview
2.	Preliminaries
3.	Example
4.	**Proposed Approach**
5.	Experiments
6.	Research Questions and Findings
7.	Conclusion and Future Work
8.	Q/A

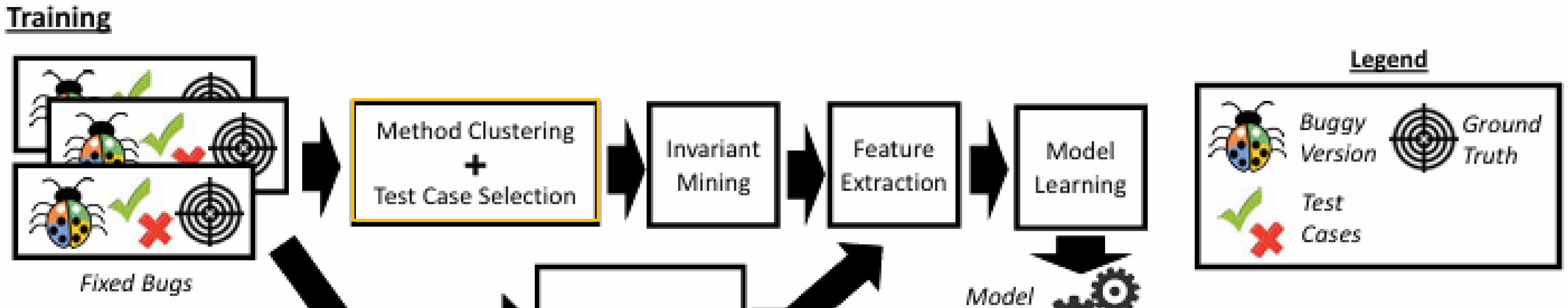
Proposed Approach

- Objective:
  - Automatically rank methods by their likelihood of containing a bug.
- Two Main Phases:
  - Training Phase: Learn a ranking model from fixed bugs.
  - Deployment Phase: Apply the model to new buggy programs.
- Key Steps in Training:
  1. Method Clustering & Test Case Selection: Reduce the search space.
  2. Invariant Mining: Extract invariants from execution traces.
  3. Feature Extraction: Compute features from invariant differences and SBFL scores.
  4. Model Learning: Train a ranking model (using rankSVM)
- Input:
  - Buggy program version
  - Failing/Passing test cases
  - Ground Truth bug location



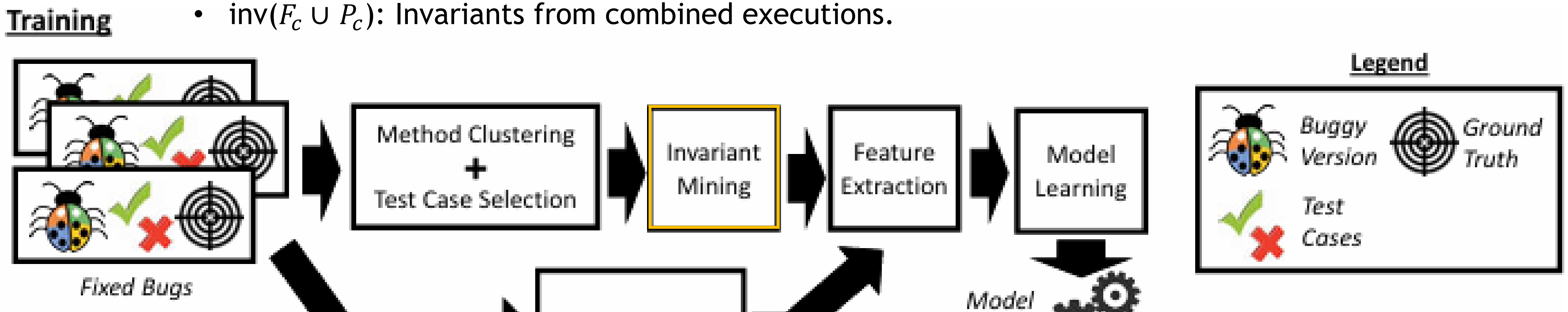
Proposed Approach (Method Clustering & Test Case Selection)

- Purpose:
  - Reduce computational cost by limiting the number of methods and tests analyzed.
- Process:
  - Exclude Irrelevant Methods: Discard methods not executed by failing tests.
- Clustering Methods:
  - Represent each method as a coverage vector (1 if a test covers it, 0 otherwise).
  - Use k-means clustering to group similar methods together.
  - Limit each cluster to a maximum size of 10 (M = 10).
- Test Case Selection:
- For each cluster, select a subset of passing tests.Ensure every method in the cluster is covered by at least 10 tests (T = 10) using a greedy selection based on test coverage.



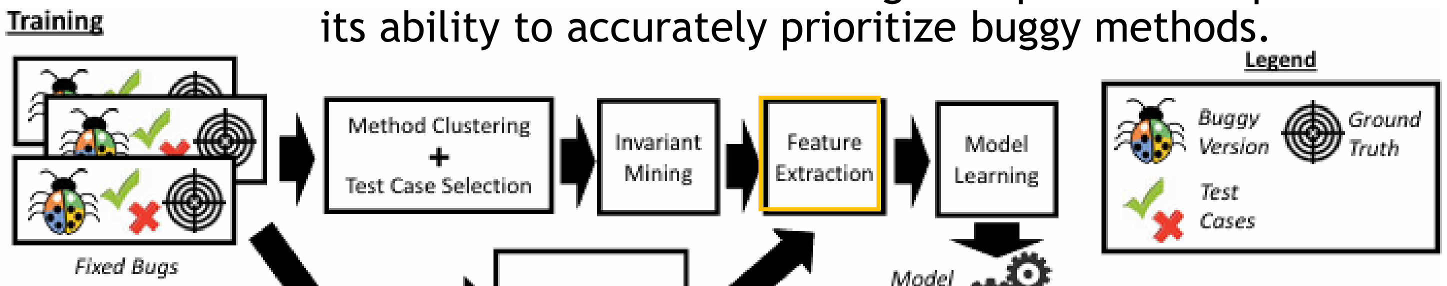
Proposed Approach (Invariant Mining)

- Objective:
  - Capture method behavior through invariants.
- Process:
  - Run test cases on each cluster to collect execution traces.
- Use three sets of executions:
  - F_c: Failing test cases.
  - P_c : Selected passing test cases.
  - $F_c \cup P_c$: Combined executions (both failing and passing).
- Tool:
  - Daikon: Automatically infers invariants at method entry/exit.
- Outputs:
  - inv(F_c): Invariants from failing tests.
  - inv(P_c): Invariants from passing tests.
  - inv($F_c \cup P_c$): Invariants from combined executions.



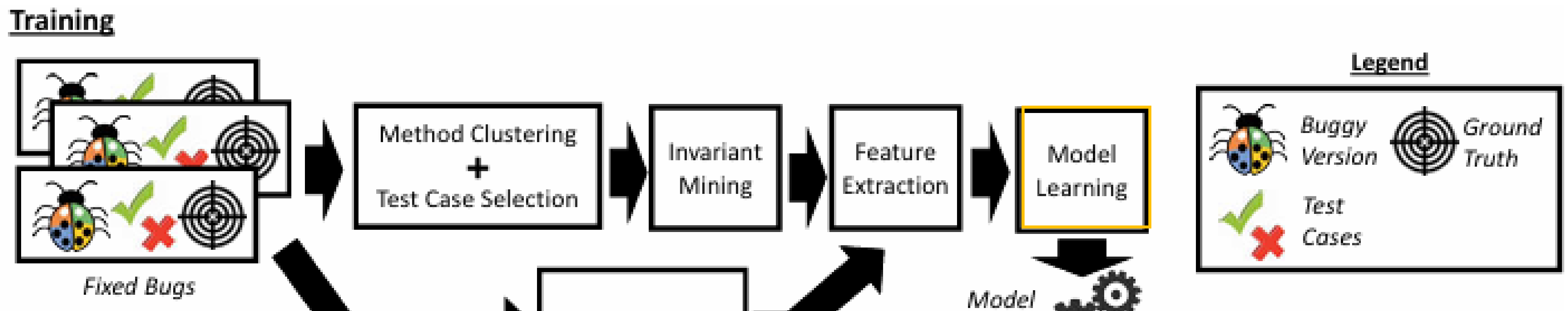
Proposed Approach (Feature Extraction)

- Objective of Feature Extraction:
  - Convert qualitative differences in program behavior into quantitative data.
  - Capture subtle changes in invariants (from passing vs. failing tests) and traditional fault localization scores.
  - Provide a rich, combined feature set that enables the ranking model to distinguish between faulty and non-faulty methods.
- Key Points:
  - Invariant Diff:
    - Compares invariants from passing tests to those from combined tests.
    - Generates 3-tuple features (e.g., [Invariant A, Invariant B, Difference Label]).
    - Abstracts detailed invariants to general types for consistency.
- Suspiciousness Scores:
  - Computes scores using multiple SBFL formulas.
  - Extracts 35 features that capture how suspicious each method is based on test coverage.
- Combined Approach:
  - Both sets of features are merged to provide comprehensive input to the learning-to-rank model, enhancing its ability to accurately prioritize buggy methods.



Proposed Approach (Model Learning)

- Feature Normalization:
  - Normalize each feature value to a range of [0, 1]
  - Purpose: Ensures all features contribute proportionately to the model.
- Model Learning:
  - Train a ranking model using rankSVM on feature vectors from fixed bugs.
  - Exclude bugs with no invariant differences.
  - Purpose: Learn a model that accurately prioritizes methods likely to be faulty.
- Deployment Phase:
  - For a new bug, extract features using the same process.
  - Apply the learned model to generate a ranked list of suspicious methods.
  - Purpose: Quickly identify the most likely bug locations in new, unseen programs.



Index

Experiments

- Empirical Evaluation:
  - Tested on 357 real bugs from 5 Java projects (Defects4J)
  - Real bugs ensure realistic debugging challenges
- Comparative Techniques & Validation:
  - Compared Savant against 12 baseline SBFL methods (e.g., ER1a, GP13, Multric, Carrot+)
  - Method-level localization; supervised methods use leave-one-out cross validation
- Parameter Settings & Environment:
  - Clustering: Maximum cluster size M = 10; Minimum test coverage T = 10
  - Tools: Daikon v5.2.84, scikit-learn 0.17.0, rankSVM (LIBSVM 1.956)
  - System: Intel® Xeon E5-2667 2.9 GHz, Linux 2.6
- Evaluation Metrics:
  - acc@n: Bugs localized within top-n positions (acc@1, @3, @5)
  - wef@n: Wasted effort (non-faulty methods inspected)
  - MAP: Mean Average Precision
  - Robustness: Experiments repeated 100 times with randomized tie-breaking

Index

No	Content
1.	Overview
2.	Preliminaries
3.	Example
4.	Proposed Approach
5.	Experiments
6.	**Research Questions and Findings**
7.	Conclusion and Future Work
8.	Q/A

Research Questions and Findings

- RQ1: Effectiveness
  - Avg. acc@1: 63.03, acc@3: 101.72, acc@5: 122 localized bugs
  - Overall MAP: 0.221
  - Best performance on Commons Lang (acc@k up to 41; MAP 0.535)
- RQ2: Comparison to Baselines
  - Outperforms 12 state-of-the-art SBFL techniques
  - ~57.73% improvement at acc@1 compared to best baselines (e.g., ER1b, GP13
- RQ3: Impact Of Feature Sets
  - Combining invariant change and suspiciousness score features yields the best results
  - Using only one type reduces effectiveness
- RQ4: Training Data Requirement
  - Performance varies only slightly with different training sizes
  - 5-fold cross validation provides the best balance
- RQ5: Efficiency
  - Average running time: 13.894 seconds overall
  - Varies by project: 1.53 sec (JFreeChart) to 31.845 sec (Closure Compiler)

Index

Conclusion

- Savant Achievements:
  - Significantly improves fault localization accuracy.
  - Reduces the candidate set using invariant mining and efficient test case selection.
  - Outperforms 12 baseline techniques in key metrics.
- Practical Impact:
  - Robust performance across various projects and training data sizes.
  - Efficient enough for practical debugging tasks.
- Future Directions:
  - Expand evaluation to more bugs and programming languages.
  - Refine invariant selection and feature integration further.

Q&A