Statistical Debugging

Ben Liblit, University of Wisconsin–Madison
Bug Isolation Architecture

Program Source

Predicates

Sampler

Compiler

Shipping Application

Statistical Debugging

Counts & 😊/😢

Top bugs with likely causes
Winnowing Down the Culprits

- 1710 counters
  - 3 × 570 call sites
- 1569 zero on all runs
  - 141 remain
- 139 nonzero on at least one successful run
- Not much left!
  - `file_exists() > 0`
  - `xreadline() == 0`
Multiple, Non-Deterministic Bugs

- Strict process of elimination won’t work
  - Can’t assume program will crash when it should
  - No single common characteristic of all failures

- Look for general correlation, not perfect prediction

- *Warning! Statistics ahead!*
Consider each predicate $P$ one at a time
   - Include inferred predicates (e.g. $\leq$, $\neq$, $\geq$)

How likely is failure when $P$ is true?
   - (technically, when $P$ is observed to be true)

Multiple bugs yield multiple bad predicates
Some Definitions

\[
F(P) = \# \text{failing runs with } |P| > 0
\]

\[
S(P) = \# \text{successful runs with } |P| > 0
\]

\[
Bad(P) = \frac{F(P)}{S(P) + F(P)}
\]
Are We Done? Not Exactly!

```c
if (f == NULL) {
    x = 0;
    *f;
}
```

$Bad(f = \text{NULL}) = 1.0$
Are We Done? Not Exactly!

```c
if (f == NULL) {
    x = 0;
    *f;
}
```

- **Predicate** ($x = 0$) is innocent bystander
- **Program** is already doomed

<table>
<thead>
<tr>
<th>Bad(f = NULL)</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad(x = 0)</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Fun With Multi-Valued Logic

- Identify unlucky sites on the doomed path

\[
\text{Context}(P) = \frac{F(P \lor \neg P)}{S(P \lor \neg P) + F(P \lor \neg P)}
\]

- Background risk of failure for reaching this site, regardless of predicate truth/falsehood
Isolate the Predictive Value of $P$

Does $P$ being true increase the chance of failure over the background rate?

$$\text{Increase}(P) = \text{Bad}(P) - \text{Context}(P)$$

- Formal correspondence to likelihood ratio testing
Increase Isolates the Predictor

if (f == NULL) {
    x = 0;
    *f;
}

\begin{align*}
\text{Increase}(f = \text{NULL}) &= 1.0 \\
\text{Increase}(x = 0) &= 0.0
\end{align*}
It Works!

…for programs with just one bug.

- Need to deal with multiple bugs
  - How many? Nobody knows!

- Redundant predictors remain a major problem

Goal: isolate a single “best” predictor for each bug, with no prior knowledge of the number of bugs.
Multiple Bugs: Some Issues

- A bug may have many redundant predictors
  - Only need one, provided it is a good one

- Bugs occur on vastly different scales
  - Predictors for common bugs may dominate, hiding predictors of less common problems
Guide to Visualization

- Multiple interesting & useful predicate metrics
- Simple visualization may help reveal trends

\[
\log(F(P) + S(P))
\]
Confidence Interval on $\text{Increase}(P)$

\[ \pm 1.96 \sqrt{\frac{\text{Bad}(P) \cdot (1 - \text{Bad}(P))}{S(P) + F(P)}} + \frac{\text{Context}(P) \cdot (1 - \text{Context}(P))}{S(P \lor \neg P) + F(P \lor \neg P)} \]

- Strictly speaking, this is slightly bogus
  - $\text{Bad}(P)$ and $\text{Context}(P)$ are not independent
  - Correct confidence interval would be larger
Bad Idea #1: Rank by $Increase(P)$

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
<th>F + S</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.065</td>
<td>0.935 ± 0.019</td>
<td>0</td>
<td>23</td>
<td>23</td>
<td><code>(* (fi + i)) -&gt; this.last_token &lt; filesbase</code></td>
</tr>
<tr>
<td></td>
<td>0.065</td>
<td>0.935 ± 0.020</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td><code>(* (fi + i)) -&gt; other.last_line == last</code></td>
</tr>
<tr>
<td></td>
<td>0.071</td>
<td>0.929 ± 0.020</td>
<td>0</td>
<td>18</td>
<td>18</td>
<td><code>(* (fi + i)) -&gt; other.last_line == filesbase</code></td>
</tr>
<tr>
<td></td>
<td>0.073</td>
<td>0.927 ± 0.020</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td><code>(* (fi + i)) -&gt; other.last_line == yy_n_chars</code></td>
</tr>
<tr>
<td></td>
<td>0.071</td>
<td>0.929 ± 0.028</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>bytes &lt;= filesbase</td>
</tr>
<tr>
<td></td>
<td>0.075</td>
<td>0.925 ± 0.022</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td><code>(* (fi + i)) -&gt; other.first_line == 2</code></td>
</tr>
<tr>
<td></td>
<td>0.076</td>
<td>0.924 ± 0.022</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td><code>(* (fi + i)) -&gt; this.first_line &lt; nid</code></td>
</tr>
<tr>
<td></td>
<td>0.077</td>
<td>0.923 ± 0.023</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td><code>(* (fi + i)) -&gt; other.last_line == yy_init</code></td>
</tr>
</tbody>
</table>

High $Increase$ but very few failing runs

These are all *sub-bug predictors*

- Each covers one special case of a larger bug

Redundancy is clearly a problem
Bad Idea #2: Rank by $F(P)$

- Many failing runs but low \textit{Increase}
- Tend to be \textit{super-bug predictors}
  - Each covers several bugs, plus lots of junk

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
<th>F + S</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22554</td>
<td>5045</td>
<td>27599</td>
<td><code>files[filesindex].language != 15</code></td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22566</td>
<td>5045</td>
<td>27611</td>
<td><code>tmp == 0</code> is FALSE</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.007 ± 0.012</td>
<td>22571</td>
<td>5045</td>
<td>27616</td>
<td><code>strcmp != 0</code></td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.007 ± 0.013</td>
<td>18894</td>
<td>4251</td>
<td>23145</td>
<td><code>tmp == 0</code> is FALSE</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.007 ± 0.013</td>
<td>18885</td>
<td>4240</td>
<td>23125</td>
<td><code>files[filesindex].language != 14</code></td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.008 ± 0.013</td>
<td>17757</td>
<td>4007</td>
<td>21764</td>
<td><code>filesindex &gt;= 25</code></td>
</tr>
<tr>
<td></td>
<td>0.177</td>
<td>0.008 ± 0.014</td>
<td>16453</td>
<td>3731</td>
<td>20184</td>
<td><code>new value of M &lt; old value of M</code></td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.261 ± 0.023</td>
<td>4800</td>
<td>3716</td>
<td>8516</td>
<td><code>config.winning_window_size != argc</code></td>
</tr>
</tbody>
</table>

2732 additional predictors follow.
A Helpful Analogy

- In the language of information retrieval
  - $Increase(P)$ has high precision, low recall
  - $F(P)$ has high recall, low precision

- Standard solution:
  - Take the harmonic mean of both
  - Rewards high scores in both dimensions
### Rank by Harmonic Mean

<table>
<thead>
<tr>
<th>Thermometer</th>
<th>Context</th>
<th>Increase</th>
<th>S</th>
<th>F</th>
<th>F + S</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1585</td>
<td>1585</td>
<td>files[filesindex].language &gt; 16</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1584</td>
<td>1584</td>
<td>strcmp &gt; 0</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1580</td>
<td>1580</td>
<td>strcmp == 0</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1577</td>
<td>1577</td>
<td>files[filesindex].language == 17</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1576</td>
<td>1576</td>
<td>tmp == 0 is TRUE</td>
</tr>
<tr>
<td></td>
<td>0.176</td>
<td>0.824 ± 0.009</td>
<td>0</td>
<td>1573</td>
<td>1573</td>
<td>strcmp &gt; 0</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
<td>774</td>
<td>775</td>
<td>(*((fi + i)))-&gt;this.last_line == 1</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.883 ± 0.012</td>
<td>1</td>
<td>776</td>
<td>777</td>
<td>(*((fi + i)))-&gt;other.last_line == yyleng</td>
</tr>
</tbody>
</table>

2732 additional predictors follow

- **Definite improvement**
  - Large increase, many failures, few or no successes
- **But redundancy is still a problem**
Redundancy Elimination

- One predictor for a bug is interesting
  - Additional predictors are a distraction
  - Want to explain each failure once

- Similar to minimum set-cover problem
  - Cover all failed runs with subset of predicates
  - Greedy selection using harmonic ranking
Simulated Iterative Bug Fixing

1. Rank all predicates under consideration
2. Select the top-ranked predicate $P$
3. Add $P$ to bug predictor list
4. Discard $P$ and all runs where $P$ was true
   - Simulates fixing the bug predicted by $P$
   - Reduces rank of similar predicates
5. Repeat until out of failures or predicates
Case Study: MOSS

- Reintroduce nine historic MOSS bugs
  - High- and low-level errors
  - Includes wrong-output bugs

- Instrument with everything we’ve got
  - Branches, returns, variable value pairs, the works

- 32,000 randomized runs at $\frac{1}{100}$ sampling
## Effectiveness of Filtering

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Total</th>
<th>Retained</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>branches</td>
<td>4170</td>
<td>18</td>
<td>0.4%</td>
</tr>
<tr>
<td>returns</td>
<td>2964</td>
<td>11</td>
<td>0.4%</td>
</tr>
<tr>
<td>value-pairs</td>
<td>195,864</td>
<td>2682</td>
<td>1.4%</td>
</tr>
</tbody>
</table>
Effectiveness of Ranking

- Five bugs: captured by branches, returns
  - Short lists, easy to scan
  - Can stop early if Bad drops down

- Two bugs: captured by value-pairs
  - Much redundancy

- Two bugs: never cause a failure
  - No failure, no problem

- One surprise bug, revealed by returns!
### Analysis of exif

<table>
<thead>
<tr>
<th>Initial</th>
<th>Effective</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Initial" /></td>
<td><img src="image2" alt="Effective" /></td>
<td>$i &lt; 0$</td>
</tr>
<tr>
<td><img src="image3" alt="Initial" /></td>
<td><img src="image4" alt="Effective" /></td>
<td>$\text{maxlen} &gt; 1900$</td>
</tr>
<tr>
<td><img src="image5" alt="Initial" /></td>
<td><img src="image6" alt="Effective" /></td>
<td>$o + s &gt; \text{buf_size}$ is TRUE</td>
</tr>
</tbody>
</table>

- 3 bug predictors from 156,476 initial predicates
- Each predicate identifies a distinct crashing bug
- All bugs found quickly using analysis results
# Analysis of Rhythmbox

<table>
<thead>
<tr>
<th>Initial</th>
<th>Effective</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>tmp is FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(mp-&gt;priv)-&gt;timer is FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(view-&gt;priv)-&gt;change_sig_queued is TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(hist-&gt;priv)-&gt;db is TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rb_playlist_manager_signals[0] &gt; 269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(db-&gt;priv)-&gt;thread_reaper_id &gt;= 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>entry == entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fn == fn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>klass &gt; klass</td>
</tr>
<tr>
<td></td>
<td></td>
<td>genre &lt; artist</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vol &lt;= (float )0 is TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(player-&gt;priv)-&gt;handling_error is TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(statusbar-&gt;priv)-&gt;library_busy is TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>shell &lt; shell</td>
</tr>
<tr>
<td></td>
<td></td>
<td>len &lt; 270</td>
</tr>
</tbody>
</table>

- 15 bug predictors from 857,384 initial predicates
- Found and fixed several crashing bugs
How Many Runs Are Needed?

<table>
<thead>
<tr>
<th></th>
<th>Failing Runs For Bug #n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1</td>
</tr>
<tr>
<td>Moss</td>
<td>18</td>
</tr>
<tr>
<td>ccrypt</td>
<td>26</td>
</tr>
<tr>
<td>bc</td>
<td>40</td>
</tr>
<tr>
<td>Rhythmbox</td>
<td>22</td>
</tr>
<tr>
<td>exif</td>
<td>28</td>
</tr>
</tbody>
</table>
Other Models, Briefly Considered

- Regularized logistic regression
  - S-shaped curve fitting

- Bipartite graphs trained with iterative voting
  - Predicates vote for runs
  - Runs assign credibility to predicates

- Predicates as random distribution pairs
  - Find predicates whose distribution parameters differ

- Random forests, decision trees, support vector machines, …
Capturing Bugs and Usage Patterns

- Borrow from natural language processing
  - Identify topics, given term-document matrix
  - Identify bugs, given feature-run matrix

- Latent semantic analysis and related models
  - Topics ⇔ bugs and usage patterns
  - Noise words ⇔ common utility code
  - Salient keywords ⇔ buggy code
Probabilistic Latent Semantic Analysis

observed data: \( \Pr(\text{pred}, \text{run}) \)

\( \approx \) \( \cdot \) \( \cdot \)

\( \Pr(\text{pred} | \text{topic}) \)

\( \cdot \) \( \cdot \) \( \Pr(\text{run} | \text{topic}) \)

topic weights
Uses of Topic Models

- Cluster runs by most probable topic
  - Failure diagnosis for multi-bug programs

- Characterize representative run for cluster
  - Failure-inducing execution profile
  - Likely execution path to guide developers

- Relate usage patterns to failure modes
  - Predict system (in)stability in scenarios of interest
Compound Predicates for Complex Bugs
“Logic, like whiskey, loses its beneficial effect when taken in too large quantities.”

Limitations of Simple Predicates

- Each predicate partitions runs into 2 sets:
  - Runs where it was true
  - Runs where it was false

- Can accurately predict bugs that match this partition

- Unfortunately, some bugs are more complex
  - Complex border between good & bad
  - Requires richer language of predicates
Motivation: Bad Pointer Errors

In function `exif_mnote_data_canon_load`:

```c
for (i = 0; i < c; i++) {
    ...
    n->count = i + 1;
    ...
    if (o + s > buf_size) return;
    ...
    n->entries[i].data = malloc(s);
    ...
}
```

Crash on later use of `n->entries[i].data`.

ptr = junk *ptr
Motivation: Bad Pointer Errors

<table>
<thead>
<tr>
<th>Kinds of Predicate</th>
<th>Best Predicate</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Only</td>
<td>new len == old len</td>
<td>0.71</td>
</tr>
<tr>
<td>Simple &amp; Compound</td>
<td>o + s &gt; buf_size ∧ offset &lt; len</td>
<td>0.94</td>
</tr>
</tbody>
</table>

- In function `exif_mnote_data_canon_load`:
  - for (i = 0; i < c; i++) {
    - ... n->count = i + 1;
    - ... if (o + s > buf_size) return;
    - ... n->entries[i].data = malloc(s);
    - ... }
  - Crash on later use of n->entries[i].data
Great! So What’s the Problem?

- Too many compound predicates
  - $2^N$ functions of $N$ simple predicates
  - $N^2$ conjunctions & disjunctions of two variables
  - $N \sim 100$ even for small applications

- Incomplete information due to sampling

- Predicates at different locations
Conservative Definition

- A conjunction $C = p_1 \land p_2$ is true in a run iff:
  - $p_1$ is true at least once and
  - $p_2$ is true at least once

- Disjunction is defined similarly

- Disadvantage:
  - $C$ may be true even if $p_1, p_2$ never true simultaneously

- Advantages:
  - Monitoring phase does not change
  - $p_1 \land p_2$ is just another predicate, inferred offline
Three-Valued Truth Tables

- For each predicate & run, three possibilities:
  1. True (at least once)
  2. Not true (and false at least once)
  3. Never observed

Conjunction: \( p_1 \land p_2 \)

<table>
<thead>
<tr>
<th>( p_2 )</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>F</td>
<td>?</td>
</tr>
</tbody>
</table>

Disjunction: \( p_1 \lor p_2 \)

<table>
<thead>
<tr>
<th>( p_2 )</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>T</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Mixed Compound & Simple Predicates

- Compute score of each conjunction & disjunction
  - $C = p_1 \land p_2$
  - $D = p_1 \lor p_2$

- Compare to scores of constituent simple predicates
  - Keep if higher score: better partition between good & bad
  - Discard if lower score: needless complexity

- Integrates easily into iterative ranking & elimination
Still Too Many

- Complexity: $N^2 \cdot R$
  - $N =$ number of simple predicates
  - $R =$ number of runs being analyzed
  - 20 minutes for $N \sim 500$, $R \sim 5,000$

- Idea: early pruning optimization

- Compute upper bound of score and discard if too low
  - “Too low” = lower than constituent simple predicates

- Reduce $O(R)$ to $O(1)$ per complex predicate
Upper Bound On Score

- **Harmonic mean**

  $$\uparrow \text{Harmonic mean} = \frac{\uparrow F(C)}{\uparrow F(C) + \downarrow S(C)} - \frac{\downarrow F(C \text{ obs})}{\downarrow F(C \text{ obs}) + \uparrow S(C \text{ obs})}$$

- **Sensitivity**

  $$\uparrow \text{Sensitivity}(C) = \frac{\uparrow \log F(C)}{\log \text{NumF}}$$

- **Upper Bound on**

  $$C = p_1 \land p_2$$
  
  - Find $$\uparrow F(C), \downarrow S(C), \downarrow F(C \text{ obs})$$ and $$\uparrow S(C \text{ obs})$$
  - In terms of corresponding counts for $$p_1, p_2$$
↑$F(C)$ and ↓$S(C)$ for conjunction

- **↑$F(C)$**: true runs completely overlap

  \[
  \text{Min}( F(p_1), F(p_2) )
  \]

- **↓$S(C)$**: true runs are disjoint

  \[
  \text{either 0 or } S(p_1)+S(p_2) - \text{NumS} \\
  \text{(whichever is maximum)}
  \]
Maximize two cases

- C = true
  - True runs of $p_1$, $p_2$ overlap

- C = false
  - False runs of $p_1$, $p_2$ are disjoint

$$\text{NumS} = \min(S(p_1), S(p_2)) + S(\neg p_1) + S(\neg p_2)$$

(whichever is minimum)
Usability

- Complex predicates can confuse programmer
  - Non-obvious relationship between constituents
  - Prefer to work with easy-to-relate predicates

- \[ \text{effort}(p_1, p_2) = \text{proximity of } p_1 \text{ and } p_2 \text{ in PDG} \]
  - PDG = CDG ∪ DDG
  - Per Cleve and Zeller [ICSE ’05]

- Fraction of entire program
  - “Usable” only if \( \text{effort} < 5\% \)
  - Somewhat arbitrary cutoff; works well in practice
Evaluation:
What kind of predicate has the top score?
Evaluation: Effectiveness of Pruning

Analysis time: from ~20 mins down to ~1 min
Evaluation: Impact of Sampling

- Conjunctions
- Simple
- Disjunctions

Application: print_tokens
Evaluation:
Usefulness Under Sparse Sampling

Application
Sampling rate = 1/100, effort < 5%