Neural Attribution for Semantic Bug-Localization in Student Programs

Rahul Gupta, Aditya Kanade, Shirish Shevade

Computer Science & Automation
Indian Institute of Science
Bangalore, India

NeurIPS 2019
Problem statement

• Bug – root cause of a program failure

• Bug-localization – significantly more difficult than bug-detection
  • Aids software developers
  • Aids programming course instructors in generating hints/feedback at scale

• Objective: To develop a data-driven, learning based bug-localization technique
  • Scope: student submissions to programming assignments
  • General idea: compare a buggy program with a reference implementation
  • Challenges
    • Finding a suitable reference implementation (same algorithm)
    • Finding bug-inducing differences in the presence of syntactic variation
Example

```c
#include <stdio.h>
int main(){
    int x1,y1,x2,y2;
    float slope;
    scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
    if(x1==x2) {
        printf("inf");
        return 0; }
    else {
        slope = (y2-y1)/(x2-x1);
        printf("%.2f\n" , slope);
    }
    return 0; }
```
Our Approach: NeuralBugLocator

Input: <Program, test>

Output: success:0, failure:1
Prediction Attribution

[Sundararajan et al., 2017]
Phase 1: Test Failure Classification

• Most existing DL techniques for programs use RNNs to model sequential encoding of programs
  • Not effective – AST is a better representation
  • We found CNNs to be more effective for this task than RNNs

• CNNs are designed to capture spatial neighbourhood information in data and are generally used with inputs having grid-like structure such as images

• We present a novel encoding of program ASTs and a tree convolutional neural network that allow efficient training on tree structured inputs
Program Encoding

AST for code snippet: int even=!((num%2);

AST Encoding as a 2-d matrix

```
[ 1  2  3  ]
[ 2  4  0  ]
[ 3  5  0  ]
[ 5  6  7  ]
```
Tree Convolutional Neural Network

Encoded program AST → Embedding layer 1 → 1 x 1 convolutions → 1 x max_nodes convolutions → Feature concatenation

1 x max_nodes convolutions → 3 x max_nodes convolutions → Feature concatenation

Test ID → Embedding layer 2 → Test ID embedding → Feature concatenation

Program embedding

Feature concatenation → Three layered fully connected neural network → Failure prediction
Background: Integrated Gradients (IG)

• When assigning credit for a prediction to a certain feature in the input, the absence of the feature is required as a baseline for comparing outcomes.

• This absence is modelled as a single baseline input on which the prediction of the neural network is “neutral” i.e., conveys a complete absence of signal

• For example, black images for object recognition networks and all-zero input embedding vectors for text based networks

• IG distributes the difference between the two outputs (corresponding to the input of interest and the baseline) to the individual input features
Phase 2: Neural Attribution for Bug-Localization

- Attribution baseline - a correct program similar to the input buggy program
- Attribution baseline as minimum cosine distance correct program
- Suspiciousness score for a line from IG assigned credit score
Experimental Setup – Dataset

• C programs written by students for an introductory programming class offered at IIT Kanpur
  • 29 diverse programming problems
  • programs with up to 450 tokens and 30 unique literals
  • 231 instructor written tests (about 8 tests per problem)
  • At least about 500 programs that pass at least 1 test and about 100 programs that pass all the tests
  • Discard programs that do not pass any tests
Training & Validation Datasets

- Generate ASTs using *pycparser*, discard the last one percentile of programs arranged in the increasing order of their AST size

- Remaining programs paired with test ids form the dataset
  - No. of examples ~ 270 K
  - max_nodes: 21
  - 5% set aside for validation
  - max_subtrees: 249

- Easy labelling – just need success/failure label as binary output
Evaluation Dataset

• Need ground truth in form of bug-locations for evaluation
  • Compare buggy submissions to their corrected versions (by the same student)
  • Select if $diff$ is lesser than five lines

• 2136 buggy programs
• 3022 buggy lines
• 7557 pairs of programs and failing test ids
Identifying Buggy Lines with *diff*

• Categorize each patch appearing in the diff into three categories
  • Insertion of correct lines
  • Deletion of buggy lines
  • Replacement of buggy lines with correct lines

• Programs with single line bug are trivial to map to test failures

• For multiline bugs
  • Create all non-trivial subsets of patches and apply to the buggy program
  • Use generated partially fixed programs to map failing tests to bug locations
Evaluation

• Phase 1 - model accuracy
  • Training: 99.9%  Validation: 96%  Evaluation: 54.5%
  • Evaluation dataset + test passing examples: 72%

• Phase 2

<table>
<thead>
<tr>
<th>Evaluation Metric</th>
<th>Localization queries</th>
<th>Bug-localization result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-10</td>
<td>Top-5</td>
</tr>
<tr>
<td>&lt;P,t&gt; pairs</td>
<td>4117</td>
<td>3134 (76.12%)</td>
</tr>
<tr>
<td>Lines</td>
<td>2071</td>
<td>1518 (73.30%)</td>
</tr>
<tr>
<td>Programs</td>
<td>1449</td>
<td>1164 (80.33%)</td>
</tr>
</tbody>
</table>

• Effective in bug-localization for programs having multiple bugs: 314/756 (42%), when reporting the top-10 suspicious lines
Faster attribution baseline search through clustering

• Searching for baseline in all the correct programs can be expensive

• Cluster all the programs using their embeddings

• For a buggy program, search for the attribution baseline only within the set of correct programs present in its cluster

• With number of clusters set to 5, clustering affects the bug-localization accuracy by less than 0.5% in every metric while reducing the cost of baseline search by a factor of 5
### Comparison with baselines

<table>
<thead>
<tr>
<th>Technique &amp; configuration</th>
<th>Bug-localization result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-10</td>
</tr>
<tr>
<td>NBL</td>
<td>1164 (80.33%)</td>
</tr>
<tr>
<td>Tarantula-1</td>
<td>964 (66.53%)</td>
</tr>
<tr>
<td>Ochiai-1</td>
<td>1130 (77.98%)</td>
</tr>
<tr>
<td>Tarantula-*</td>
<td>1141 (78.74%)</td>
</tr>
<tr>
<td>Ochiai-*</td>
<td>1151 (79.43%)</td>
</tr>
<tr>
<td>Diff-based</td>
<td>623 (43.00%)</td>
</tr>
</tbody>
</table>

Tarantula [Jones et al., 2001], Ochiai [Abreu et al., 2006]
Qualitative Evaluation

- NeuralBugLocator localized all kinds of bugs appearing in the evaluation dataset
  - wrong assignments
  - conditions
  - for-loops
  - memory allocations
  - output formatting
  - incorrectly reading program inputs
  - missing code
Wrong Assignment/Type Narrowing

```c
#include <stdio.h>
int main(){
    int x1,y1,x2,y2;
    float slope;
    scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
    if(x1==x2) {
        printf("inf");
        return 0; 
    } else {
        slope=((y2-y1)/(x2-x1)); \ \ suspiciousness score: 0.0028934027
        printf("%.2f\n", slope);
    }
    return 0; }
```
Wrong Input and Output Formatting

```c
#include <stdio.h>

int main(){
    int n,i;
    char c;
    scanf("%d",&n); // suspiciousness score: 0.0007697232
    for(i=0;i<n;i++) {
        scanf("%c",&c);
        if (c=='a'|| c=='e' || c=='i' || c=='o'|| c=='u') {
            printf("Special");
            printf("\n%d",i); // suspiciousness score: 0.00045288168
            break; } } }
    if(i==n)
    printf("Normal");
    return 0; }
```
```c
#include <stdio.h>

int main() {
    int n, i;
    char a[100];
    char b;
    int flag = 0;
    scanf("%d", &n);
    for (i = 0; i < n; i = i + 1) {
        scanf("%c", &b);
        if ((b == 'a') || (b == 'e') || (b == 'i') || (b == 'o') || (b == 'u')) \ suspiciousness
            score: 0.0015987115
            flag = 1; }
    if (flag == 1) {
        printf("Special"); }
    else {
        printf("Normal"); }
    return 0; }
```
#include <stdio.h>

int rot(int [], int, int);

int main() {
    int n, d, i;
    scanf("%d\n", &n);
    int arr[n];
    for (i=0; i<n; i++) {
        scanf("%d ", &arr[i]);
    }
    scanf("\n%d", &d);
    rot(arr, n, d);
    return 0;
}

int rot(int arr[], int n, int d) {
    int j, k;
    for (j=d+1; j<n; j++) {
        printf("%d ", arr[j]);
    }
    for (k=0; k<=d; k++) {
        printf("%d ", arr[k]);
    }
    return 0;
}
Limitations & Future Work

• Can be used only in a restricted setting
  • Requires training data including a reference implementation

• Model accuracy
  • Wrong classification of buggy programs
  • Wrong classification of correct programs

• Idea is general and benefits from improvements in underlying techniques

• Evaluation in the setting of regression testing

• Extension to achieve neural program repair
Conclusion

• A novel encoding of program ASTs and a tree convolutional neural network that allow efficient batch training for arbitrarily shaped trees

• First deep learning based general technique for semantic bug-localization in programs. Also introduces prediction attribution in the context of programs

• Automated labelling of training data. Does not require actual bug-locations as ground truth

• Competitive with expert-designed bug-localization algorithms. Successfully localized a wide variety of semantic bugs, including wrong conditionals, assignments, output formatting and memory allocation, etc.

https://bitbucket.org/iiscseal/NBL
Acknowledgements

• Prof. Amey Karkare and his research group from IIT-Kanpur for dataset
• Sonata Software for partial funding of this work
• NVIDIA for a GPU grant