Spectrum Based Fault Localization using Graph Neural Networks

CS396A
Siddharth Satyam, Honey Nikam

November 28, 2021

Abstract

Fault localization methods have been widely studied and various learning and spectrum based methods have been employed in prior literature. Traditional techniques have used evaluation metrics based on Spectrum based Fault Localization (SBFL) such as Ochiai, Tarantula, DStar, Jaccard etc. which provide suspiciousness scores for each component that can be used to detect bugs. However, it is possible that bugs are not executed by a single failing test case. Since the involvement in failing test case significantly affects localizability of a bug with SBFL techniques, this limits the effectiveness of SBFL. In this study, we have employed Graph Neural Networks(GNN) to create prediction scores for fault localization. The graph structure is created by treating the spectrum data to be a network of test cases and components and Deep Graph Library(DGL) has been used to implement the GNN. Furthermore, we have compared the performance with the SBFL metric Ochiai.

Keywords: Graph Neural Networks, Spectrum Based Fault Localization

1 Introduction

In recent times, manual software debugging costs a lot of time as well as effort. Automated techniques have thus become indispensable to avoid the associated manual labour. Fault localization has been a topic of study, and several techniques have been employed in prior literature. The most commonly used techniques employ methodologies that produce ranks based on suspiciousness scores for the program components. The developers can then manually work on the components in the order of their ranks to find the bug in the program. Among the used techniques, Spectrum based Fault Localization has been widely adopted due to its simplicity and lightweightness. These metrics such as Ochiai, Tarantula, Dstar, Jaccard etc. use statistical analysis on the coverage data of failed/passed tests. The statistical analysis relies on the intuition that a failing test case may correspond to a buggy component. The major limitation of the technique is that a failing test case may execute a component that is not buggy and a buggy component might coincidentally correspond to a passing component. To bridge the limitations, Mutation-based Fault Localization (MBFL) was widely used. Moreover, machine learning techniques have been employed such as Learning-to-rank [1], which is a supervised machine learning algorithm that uses the suspiciousness scores as features for improved fault localization. Recently, a technique employing machine learning, DeepFL was introduced [1]. It takes suspiciousness-based features from the fault localization area (including both SBFL and MBFL), fault-proneness-based features and textual-similarity-based features from the information retrieval area. These methods have significantly improved over the conventional SBFL metrics and encouraged further research in the area. In our work, we have used Graph Neural Networks(GNN) to attempt the Spectrum-based fault localization problem. We treat the test cases and components as a network of nodes with directed edges that correspond to the execution of components by test cases.

1.1 Spectrum

A spectrum consists of an activity matrix and a corresponding error vector. The activity matrix consists of rows corresponding to test cases and columns corresponding to components. An element in the matrix describes which test case executes the specific program component. An error vector, additionally, provides the information about whether the test case is passing or failing. Fig.1 shows a spectrum consisting of an activity matrix and error vector.

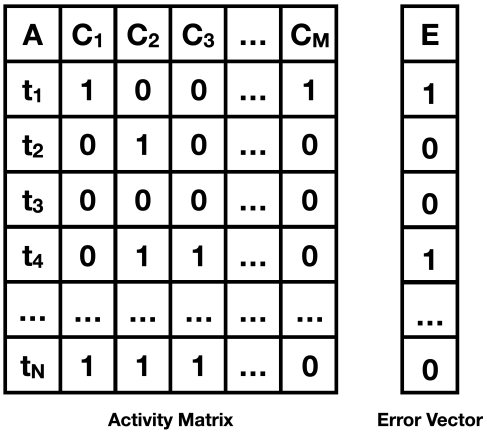

Figure 1: A spectrum consisting of an activity matrix and error vector

1.2 Metric-based SBFL techniques

These techniques use ranking metric formulas to produce the suspiciousness scores. To determine correlation between program components and test case results, they use program spectrum information as input. Ochiai, a widely used SBFL metric, is described as follows:
e_f = No. of failed tests that execute the component.
e_p = No. of passed tests that execute the component.
n_f = No. of failed tests that do not execute the component.
n_p = No. of passed tests that do not execute the component.

$$Ochiai(component) = \frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}} \tag{1}$$

1.3 Graph Neural Network

GNNs are neural networks that encode the dependence of nodes from neighboring nodes in a graph[2]. The dependence can be quantified by an embedding vector of a node which carries information through message passing iterations from its neighbors. An embedding vector is a result of graph representation learning [2], that represents the graph nodes and edges as low dimensional vectors that can be used as features in learning-based procedures. The message passing process comprises of aggregation and an additional update operation that results in an embedding that can be further used as features in a neural network. The simple aggregation function can involve averaging the neighbor messages. A neural network could then be applied to make the function complex. The process can be given by equation (2):

$$h_v^k = \sigma(W_k \cdot \sum_{u \in N(v)} \frac{h_u^{k-1}}{|N(v)|} + B_k \cdot h_v^{k-1}) \tag{2}$$

where h_v^k is the k^{th} layer embedding of a node v, σ is a non-linear function such as ReLU or tanh, $N(v)$ is the neighbor set of node v, W_k is the weight associated with the neural network and h_v^{k-1} is the previous layer embedding of v.

2 Methodology

Usually, GNN models assign multidimensional vectors, or embeddings $\in \mathbb{R}^2$ to vertices. These embeddings are then refined using information from neighboring nodes through a number of message passing iterations [4]. The adjacency information controls which are the valid incoming messages for a given vertex, these filtered messages undergo an aggregating function and finally a neural network such as feed-forward network receives the aggregated messages and computes the embedding update for the given vertex.

2.1 Setting up the Graph

In our implementation, the graph is made of two types of nodes, components and test cases. If a test case executes a particular component, then two directed edges exist between the two nodes, one from the component node to the test case node and another in the opposite direction as shown in Fig.2. Through the error vector we have information about which test cases pass and which ones fail. This information is encoded as node embeddings. For a failing test case, it's embedding is a 1×5 vector where each value is 1. Similarly for a passing test case, the embedding is a 1×5 vector of zeros.

2.2 Node Aggregation

The graph neural network structure allows for exchange of information among the different nodes. For example, if a test case T is failing and is executing components A and B, the

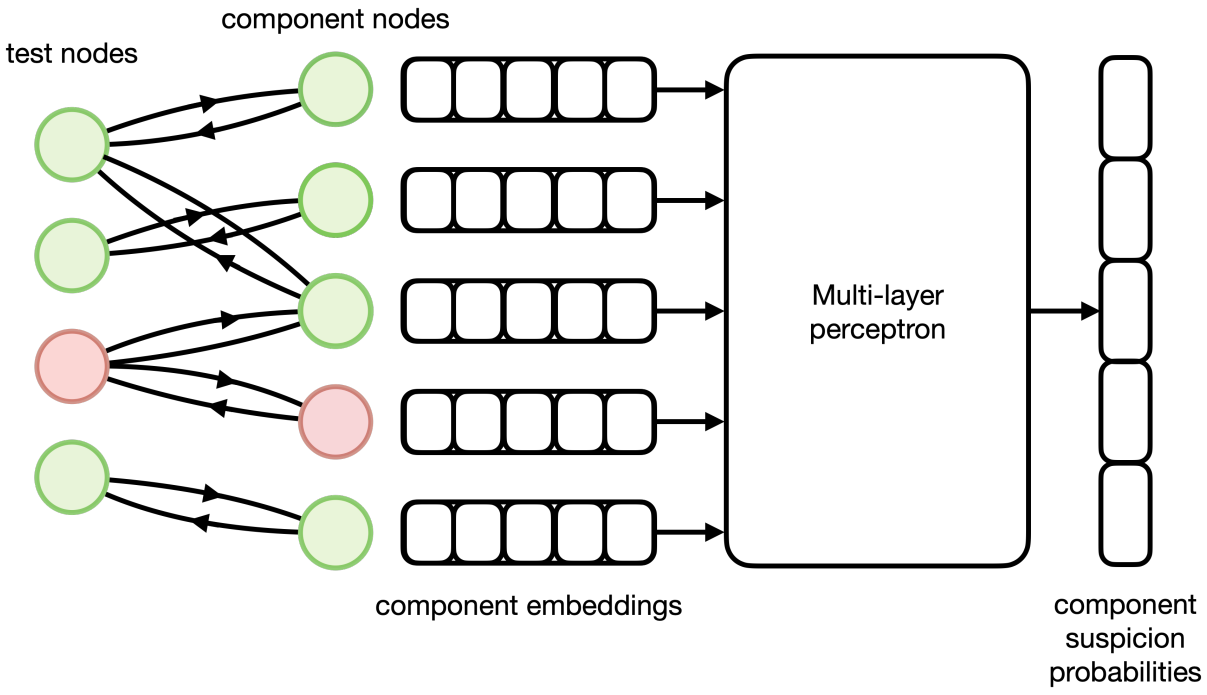

Figure 2: Component embeddings are computed using aggregation after multiple message passing iterations. These embeddings are fed into an MLP to give component suspicion probabilites

components must receive the information that the test case T is failing. This information propagates through the edges that exist between the nodes via node aggregation. When component node A aggregates, it pulls information from its neighboring nodes (in our case, test cases that are executing it) and updates its own node embedding. In our implementation we have used a simple node aggregation function which sums the neighboring node embeddings and averages them to update the current node embedding. There is a traversal order which can be specified for node aggregation. In our case, we first have all node components aggregate information at once, then we have all test case components aggregate information, followed again by node components aggregating embeddings. Multiple passes between the test cases and components allows for information to flow between unconnected nodes as well.

2.3 Deep Graph Library

Deep Graph Library (DGL) is a Python package built for easy implementation of graph neural network model family. We use the *dgl.heterograph* function to build a graph from the activity matrix and assign node embeddings for the different nodes. For passing messages to and fro, we use the built in function *prop_nodes*, which takes in different inputs, the first being the traversal order. The traversal order is specified by the *nodes_generator*. It generates node frontiers, which is a list or a tensor of nodes. The nodes in the same frontier will be triggered together, while nodes in different frontiers will be triggered according to the generating order. We use the dgl built-in function 'mean' that aggregates messages by mean.

2.4 The Multi-layer Perceptron

The updated embeddings of component nodes after multiple message passes serve as the training input to a multi-layer perceptron. The output are the actual component labels, i.e., the information if a component is buggy or not. Component label is a vector of size $num\_components \times 1$, where the value is 1 if the i^{th} component is buggy and 0 if the i^{th} component is not buggy. The MLP consists of one hidden layer. A vector of size 1×5 (component node embedding) is given as the input. This embedding is multiplied by a 5×5 weight matrix to give a 1×5 hidden layer. The hidden layer is further multiplied by a 5×1 weight matrix to give 1×1 output. A sigmoid function is applied to this output to give a logit probability of how buggy a component is. The closer the value is to 1 the more likely it is to be buggy.

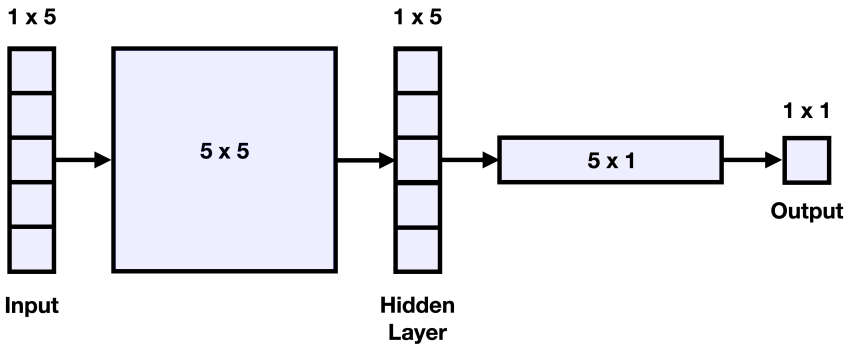

Figure 4: The MLP accepts an embedding vector and generates logit probabilities

Algorithm 1 Pseudocode

{create adjacency matrix from spectrum}
$adj[i,j] \leftarrow 1\ iff\ (\exists e \in E | e = (v_i, v_j)) | \forall v_i \in \mathbb{T}, v_j \in \mathbb{C}$

Create a dgl graph from the adjacency matrix

{assign initial test embeddings}
if $test\_label = 0$ **then**
  $test\_embedding \leftarrow [0,0,0,0,0]$
else
  $test\_embedding \leftarrow [1,1,1,1,1]$
end if

{randomly initialize component embeddings}
$component\_embedding \leftarrow \mathcal{U}(0,1)$

Perform message passing iterations to update embeddings using prop_nodes
message_passing(traversal order, message function, aggregation function)

{use an MLP to generate logit probabilities}
$logit \leftarrow \mathcal{MLP}(component\_embeddings)$

3 Results and Discussion

3.1 Node Embeddings

The results show a pattern in the computed component embeddings after several message passes. We considered a graph of 8 nodes with 3 test cases and 5 components. As shown in Fig 4., test case 1 is a failing test case and test case 0, 2 are passing test cases. First, the components pull information from neighboring nodes and aggregate, then the test cases pull information and perform aggregation followed again by the component nodes. We see that components 3 and 4 are connected to the failing test cases and hence are more likely to be buggy. Since 3 is only connected to a failing test case while 4 is connected to passing test case as well, it is also intuitive that 3 is more likely to be buggy than 4. We see that this is reflected in the node embeddings of the components. The node embedding of component 3 generated after several message passes $[0.7500, 0.7500, 0.7500, 0.7500, 0.7500]$ is greater than that of 4 $[0.4583, 0.4583, 0.4583, 0.4583, 0.4583]$. The embeddings of component 5, 6, 7 are $[0.0000, 0.0000, 0.0000, 0.0000, 0.0000]$, $[0.1667, 0.1667, 0.1667, 0.1667, 0.1667]$ and $[0.1667, 0.1667, 0.1667, 0.1667, 0.1667]$ respectively. Hence we see that a pattern is generated in the embeddings where a value closer to 1 reflects that the component is likely to be buggy whereas a value closer to 0 indicates that the component is likely to be non-buggy. The pattern reflected in the component node embedding were then used as inputs in the MLP.

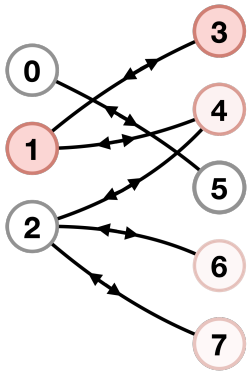

Figure 5: A graph structure with 3 test case nodes and 5 component nodes. The color intensity of component nodes represents their suspiciousness as result of message passing iteration.

3.2 Learning from Imbalanced Data

We performed message aggregation and trained our model to predict suspicion probabilities on a sample training data. Our initial input data had 33 test cases and 193 components. Out of the 193 components, 1 was a buggy component. Since our input data was imbalanced, i.e. the ratio of buggy components to non-buggy components was high, our model could not locate the buggy component and assigned values close to zero to all components. This gave us a high accuracy since only 1 out of 193 components was buggy.

3.3 Oversampling

A way to circumvent the problem of imbalanced data is oversampling. Oversampling the minority class, which in our case is buggy components can make the data less biased. In our initial data the 67^{th} component was buggy. We created a new input data in which every even column (indexed from 0) in the activity matrix was the same as the initial input activity matrix, and every odd column was the same as the 67^{th} component column of the initial activity matrix. Hence our new input activity matrix contains alternating buggy and non-buggy, i.e. $193 \times 2 = 386$ components out of which half are buggy. The model assigns a logit probability close to 0 to the non-buggy components and close to 0.9 to the buggy components. Hence we see that the model performs well on the training set. However the performance of the model on a random testing data is still subpar.

C_1	8.1462e-15
C_2	9.9029e-01
C_3	8.1462e-15
C_4	9.9029e-01
C_5	3.2526e-08
C_6	9.9029e-01
⋮	⋮
C_{385}	1.0826e-09
C_{386}	9.9029e-01

C_1	0.0
C_2	0.707106
C_3	0.0
C_4	0.707106
C_5	0.389249
C_6	0.707106
⋮	⋮
C_{385}	0.0
C_{386}	0.707106

Figure 6: The logit probabilities(left) and Ochiai scores(right)

4 Future Works

To make the data balanced, we alternated one single buggy-component column. It is possible that the model has memorized this particular column and hence outputs it as buggy. It is also possible that the model has learned the alternate pattern that exists in the input data and accordingly outputs an alternating output of values close to 0 and 0.9. In either case, there is a possiblity that our model is not learning from the computed component embeddings but from some patterns that have coincidentally resulted because of modifying the input data. Because of these factors, the high accuracy of the predicted data can be misleading. To make the data more randomized, we aim to work with an input data which has multiple buggy components. We will randomly oversample these component columns to create a non-biased activity matrix. We hope that this will help the model to learn using the actual component embeddings and give us better results. In the present model we train the parameters of the MLP. However the GNN does not have any trainable parameters as we use a simple mean function for aggregation. The training accuracy can improve significantly if trainable parameters are introduced in the GNN by using a neural network based aggregation function such as Graph Convolutional Networks(GCN) or Graph Attention Networks(GAN) .

References

[1] Li, Xia, Li, Wei, Zhang, Yuqun, and Zhang, Lingming.. "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization". ACM SIGSOFT International Symposium on Software Testing and Analysis. https://doi.org/10.1145/3293882.3330574.

[2] Jie Zhou et.al, Graph neural networks: A review of methods and applications, AI Open, Vol. 1, 2020, pp 57-81, ISSN 2666-6510, doi: https://doi.org/10.1016/j.aiopen.2021.01.001.

[3] Souza, Higor Chaim, Marcos Kon, Fabio. (2016). Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges.[online]

[4] Lemos, Henrique Prates, Marcelo Avelar, Pedro Lamb, Luís. (2019). Graph Colouring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems. 879-885. doi: 10.1109/ICTAI.2019.00125.

[5] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn and D. Lo, "A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System," 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 114-125, doi: 10.1109/QRS.2017.22.