Embedding Semantic Graph using Node Distance for Malware Detection

Authors: Luxi Wang, Zhen Zhang

Abstract

To effectively learn the semantic difference between malware and malicious Android applications, it is important to have use a good embedding technique the best reflect the semantic features in application. In this case, we are using the graph-based semantic representation from SeGuard. The goal of this project is to explore more advanced featurization techniques for embedding semantic graphs into vectors. The current approach is to use the concatenation of the node one-hot vector and edge one-hot vector. However, this approach does not consider the important topological information about the **difference in distance between nodes** that are not directly connected. In this project, we are seeking a solution that concatenates node-one-hot vector, edge-one-hot vector and **node distance vector** as the final feature vector.

Specifically, we are proposing a more sophisticated distance calculated using Node2Vec. Node2vec maps each node to a vector in a given dimension that embeds the information about the node's neighborhood. For example, in our project, two closely adjacent nodes should have a small Cartesian distance between their node2vec embeddings. After transforming all the nodes in a graph into vectors, we can calculate the Cartesian distance between each node vector, and use the concatenation of the distance vector and one-hot vectors as the feature vector for the graph. The advantage of this method is that it better quantifies the relationship between two nodes, and we think that model will better know which certain API pattern identifies which type of Android malware. While node2vec is a general and flexible tool, we need to adjust the (hyper-)parameters so that it works well for our purpose. The possible related parameters are: (1) dimension, (2) length of walk, (3) number of walks, (4) return hyper-parameter and (5) input hyper-parameter. By perturbing these parameters, the distance calculated from the embeddings represents the distance on the original graph. These parameters determines if the distance between nodes makes sense. In this work we will empirically find a parameter set that results in embeddings that gives a more accurate classification.

Main Takeaways

- Random Forest classifier cannot distinguish the difference between nodes that are closely connected (few nodes in between) or loosely connected (a lot of nodes in between) using only node and edge one hot encodings.
- Using a node embedding tool, Node2Vec, we add a distance vector to featurization, which makes an explicit difference between nodes that close to each other and nodes that are far away. With this featurization, we are hoping to find some patterns of malicious behavior in Android malware.

Overview

Distances in Semantic Graph

> *(Definition)* ***Semantic Graph:*** *A graph in which each node represents a library method (API), and the edge from one node u to another node v represents v depends on u, e.g. control-flow dependency or data-flow dependency.*

The most intuitive way to represent a graph is to use the node-one-hot vector and edge-one-hot vector, and we call this approach as "previous approach". When we feed this kind of vectors in a Random Forest estimator, the estimator cannot tell the difference between the following simplified semantic-graph based on call-graph (Figure 1):

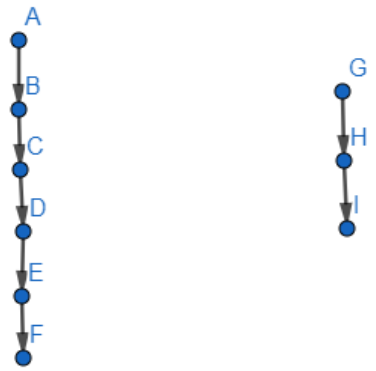

Figure 1: Random Forest Tree cannot distinguish the difference using only edge one-hot encoding

Clearly, method `A` transitively called method `F` and method `G` transitively called method `I`, but there is a fundamental difference in these two situations.
Method `A` and method `F` are not strongly correlated, that is to say, calling method `A` does not necessarily guarantee that method `F` is called. For example, in the code snippet below, method `B` only calls method `C` if a certain condition is satisfied. Thus, method `F` will not be called in this situation.
However, the situation with `G` and `I` are very different. It is far more likely at runtime (since static call-graph is imprecise) that if `G` is called, then `I` is called too. You could argue that `G` might also call `I` in a certain condition, but the less intermediate nodes in between, the less likely that this situation shall happen. The design of our featurization is based on this assumption.

```
def A():
  ...
  B(a)
  ...
def B(a):
  ...
  if(...):
    C(b)
  ...
...
```

```
def E(d):
  F(e)
  ...
```

```
def G():
  ...
  H()
def H():
  ...
  I()
def I():
```

If we use the the previous featurization solution, Random Forest estimator cannot explicitly distinguish between these conditions. Thus, we propose to add a set of new features to the original ones based on node distance.
The node distance vector represents the distance of every node to every other node in the data set on this graph. In the context of an Android application, it means how strongly two methods are correlated. In the example above, we can think of the distance between method `A` and method `F` is *long*, and the distance between method `G` and method `I` is *short*.
Android applications of the same type tend to have similar patterns, in which a specific group of methods have close connections with another group of methods. We believe the node distance vector can capture these patterns, so that the downstream machine learning classifiers can take advantage of these features.
The naive way to implement distance between nodes would be calculating minimum number of edges between them (e.g. using Dijkstra algorithm). In this project, we propose to use node2vec, a novel machine learning based solution, to calculate the embedding for the nodes, and then calculate the Cartesian distance between nodes.

Node Embedding using Node2Vec

Node2vec is a node embedding tool. It generates traces from random walks and then uses a word embedding tool, word2vec, to generate the embedding of traces. For example, figure 2 represents a semantic graph and one possible random walk starting from method `A` . The solid lines represent the path of the walk, and the dash lines represent existing edges that are not visited by the walk. After generating the random walk constrained by some parameters, node2vec feeds the walk: `(['A', 'C', 'G', 'I', 'D', 'E', 'F'])` to word2vec as if it was a sentence.

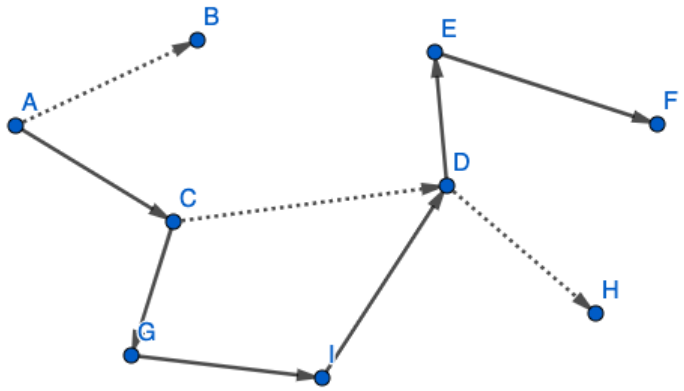

Figure 2. A possible random walk on a semantic graph

Node2vec is flexible in a way which allows us to discover if a certain settings should improve the performance of the model. One can adjust the parameters to control the walk. We selected 5 relevant parameters:

- **dimension**: the dimension of the embedding vector, which determines the maximum amount of features that can be encoded
- **walk length**: determines how many steps that a random walk should take. The longer the length, with other conditions unchanged, the more likely that this walk is going to visit nodes that are far away
- **number of walks**: number of walks starting from per node, the more random walks are generated, the less the bias and randomness is involved
- **return parameter** (p): determines how often that a walk is going to return to the node it just visited
- **in-out-parameter**(q): determines if the walk tends to stay local or global.

We normalize the distance between 0.0 to 1.0 to make it comparable with distances in other graphs. To be more specific, we divide every node distance by the maximum distance in the graph. Note that if a node does not exist in this graph, then the distances involving this node is set to 2.0, which means that the connection between APIs that are not used is the weakest.

Node2Vec on Simple Graphs

Node2Vec algorithm is stochastic: it might generate different embedding even using the same graph and same hyper-parameters. We have tested this behavior on one simple graph in Figure 3:

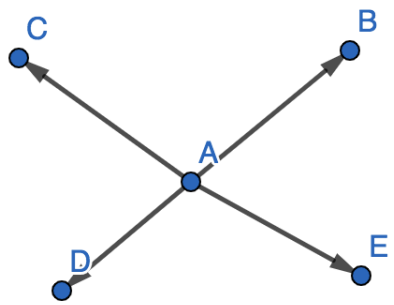

Figure 3. A simple graph

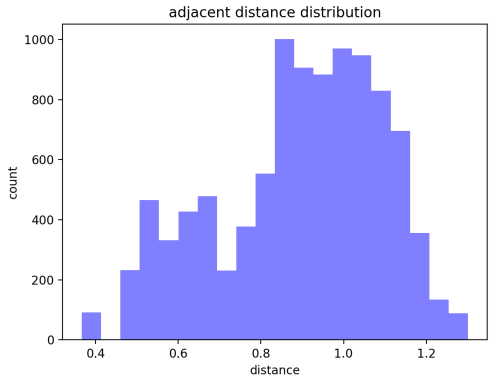

Figure 4. sample 10000 times of node2vec distance between A, B in Figure 3

If we use Node2Vec to calculate $d(A, B)$ for 10000 times, we get a different value every time. But it is clearly centered around 1.

Another graph we have tested is a single chain from node 0 (n_0) to node 20 (n_{20}) . That is, node 0 is connected to node 1, node 1 is connected to node 2, and so on until node 20.

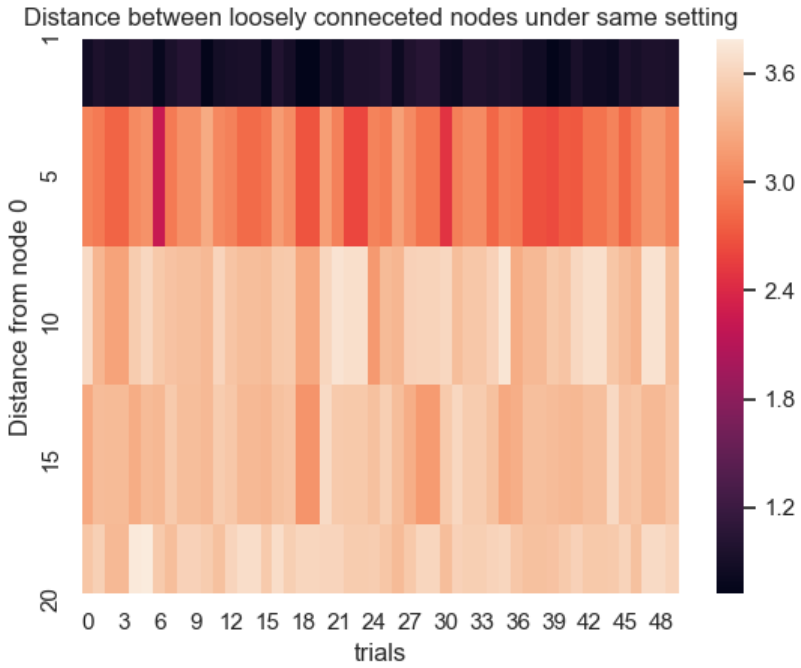

Figure 5. Distance between loosely connected nodes.

Note that, here we are comparing two kinds of distances: (1) simple graph-theory distance and (2) Cartesian distance using Node2Vec. We define *simple graph-theory* as the minimum number of edges in a path between two nodes on the graph. For example, node 0 and node 1 has one edge in between, their simple graph-theory distance is 1. Similarly, the simple graph-theory distance between node 0 and node 20 is 20. Figure 5 illustrates the relationship between simple graph-theory and Cartesian distance using Node2Vec. The x-axis represents different trials conducted in the

experiment, and the y-axis represents the simple graph-theory. The color in each block represents the corresponding Node2Vec Cartesian distance.
Although the distance is slightly different in each trial, the general pattern is observable: $d(n_0, n_1)$ is the smallest (darkest), $d(n_0, n_5)$ is in between, and if the nodes are too far away from node 0, the algorithm cannot distinguish them.

Evaluation

Data Preparation

The original dataset contains 308 graphs in total (see table below):

label	samples
benign	71
backdoor	6
phishing	2
warn_click_fraud	5
block_phishing	21
warn_spyware	32
warn_hostile_downloader	21
warn_sms_fraud	14
block_trojan	3
warn_trojan	19
block_hostile_downloader	2
warn_toll_fraud	9
warn_commercial_spyware	1
warn_phishing	1
block_backdoor	1
warn_privilege_escalation	1
warn_backdoor	6
downloader	4
trojan	2
spyware	4
jssmsers	1
Lotoor	4
gooddroid	34
ExploitLinuxLotoor	7
TrojanSMS	3

ransomware	29
sms	5

Every label that is not 'benign' implies that is is a malware.

There are some labels in the dataset that have significantly small number of graphs. We merged some labels in the dataset and removed the ones that have only 1 or 2 instances. Below is the relabelled version of the original dataset:

label	samples
sms	23
ransomware	29
benign	105
backdoor	24
downloader	27
phishing	24
trojan	24
spyware	37

Node2Vec Hyper-Parameter Selection

The embedding generated by Node2Vec can be adjusted by several parameters. We have studied 5 relevant parameters: node dimensions, length of walk, number of walk, return parameter and in-out parameter.
Our goal is to select a set (or several sets) of parameters that best reflects the actual distribution of the distances between nodes. Node2Vec embeds nodes by feeding random walks to Word2Vec, and Word2vec embeds information of words that appear in the same sentence. For example, if the sentence pass in is 'the bear is brown'. The Cartesian distance between 'bear' and 'brown' is small, indicating those words are correlated. Hence, we want to generate random walks that mostly involve the nodes around it, and exclude the nodes that are far away. According to the description in the Node2Vec paper, to force the random walk return to the previously visited nodes, set the $p < max(q, 1)$, and to make sure the random walk stays locally, set $q < 1$. Thus, we set $q = 0.2$ and $p = 0.5$.
Number of walk reduces the bias of random walk and enables different walks. Using the example from figure 3, if a random walk starts at `A`, the number of walk has to be at least 5 so it can explore all different paths starting from `A`. We set this number to be 30. We believe this number explores every branch from any single node, and less the bias due to the randomness. As for the length of walk, we used 15, an empirical number from the dataset.
Below are the graphs showing the effect of dimensions on classification accuracy.

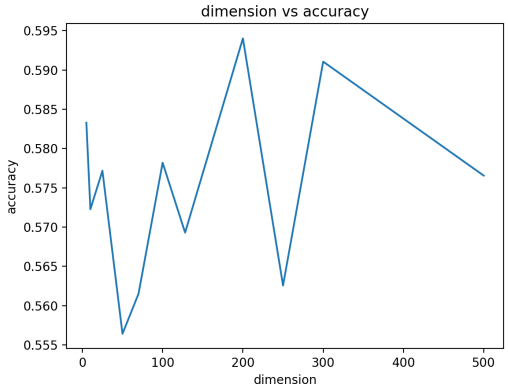

Figure 6. The effect of dimension on classification accuracy

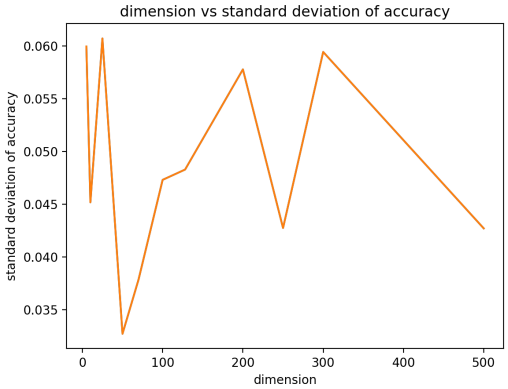

Figure 7. The effect of dimension on classification accuracy standard deviation

The effect of dimension have relatively small effect on the accuracy. Thus, in the following experiment, we just use dimension = 25.

Validating the New Featurization Approach

Since we are not sure that our labels are 100% correct, we want to create similar graphs based on the existing graphs to test our featurization. We perturb some parameters in Node2Vec in a small range which slightly changes the embedding. If our featurization is correct, the featurized vector of the same graph using slightly different parameters should be very similar.

Here is our synthesis method based on parameter tuning. There are two parameters in Node2Vec that can adjust how the algorithm explores the graph: the return parameter p and in-out parameter q. Changing the parameters results in a change of the embedding of the nodes, and thus will change the result for node distance calculation, but Node2Vec with different parameter sets are still exploring the same graph (*ground-truth*), so we expect the estimator to classify them as the same.

By adding small (with in ± 0.15) disturbance to those p and q, we are able to produce multiple embedding for the nodes from the same graph. The two figures below illustrate the effect of changing parameters on one particular node in a graph.

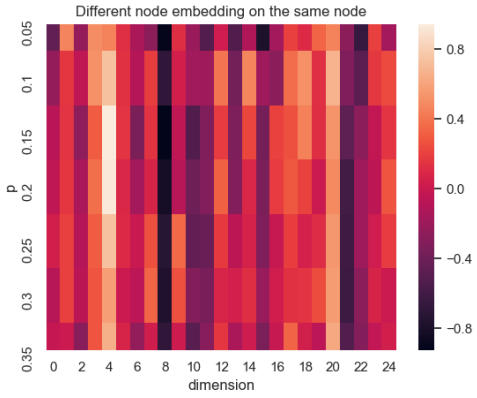

Figure 8. The effect of small disturbance of p on node embedding

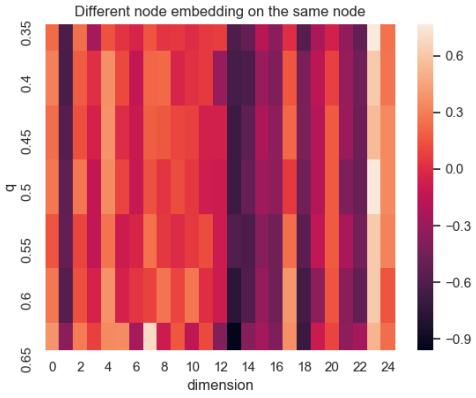

Figure 9. The effect of small disturbance of q on node embedding

Note: We are using the parameter set discussed in the previous section.

The x axis represents the dimension of the node, and the y axis represents the value we are using for p (or q). The color in each block is illustrated on the bar on the right hand side. The more similar the colors are, the more similar the numeric values are. As illustrated in the graph, with in the same column, a change in relevant parameters result in a subtle change, which changes the graph feature in a small scale. If our data makes sense, that the graph is labelled correctly, the results for classification on graph with slightly different embedding should remain the same.
Below are the heap maps generated for one sample. Figure 10 changed p and figure 11 changed q. We sample the value of q in [0.22, 0.28] and removed all the 2.0 distances, because they do not change with the change of the parameters.

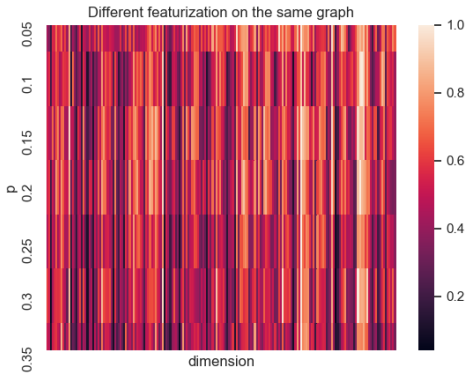

Figure 10. Effect of changing p on graph featurization

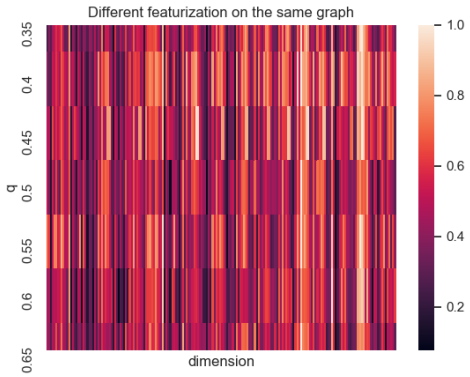

Figure 11. Effect of changing q on graph featurization

Each column of the graph represents the distance of same pair of nodes in the original graph with slightly different perturbation of p and q. The color represents the numerical value of the distance. We can see that with in a column, the change of color is subtle. A small change in the parameters results in similar featurizations. Thus, the featurization is correct.

Comparison between previous method and new method on binary and multivariate classification

In this experiment, we run binary and multivariate classification on different data set size. We draw 50, 100, 150, 200, 250, 314 samples from our data set and examine the cross-validation accuracy at each data size for each method.
In binary classification, we relabel the data with ‘malicious’ and ‘benign’.

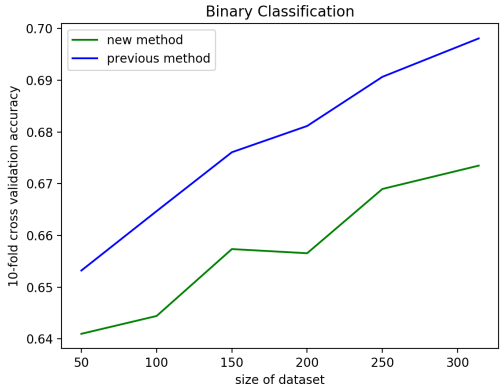

Figure 12. Size of dataset vs accuracy in binary classification

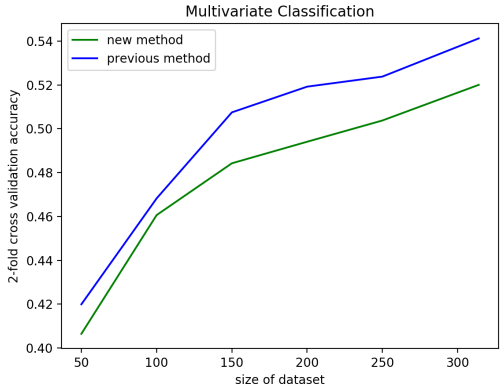

Figure 13. Size of dataset vs accuracy in binary classification

In multivariate classification, we use the labels introduced in the data preparation.

The convergence is not obvious when the size of the dataset is small. Based on the assumption that removing one node should not affect the category of the graph, we synthesized a larger dataset based on the original one using the following method: for every graph in the original dataset, randomly delete a node from the graph, and repeat for 30 times. We use the new 9000 graphs as our synthetic dataset, and run the same experiments with more samples selected.

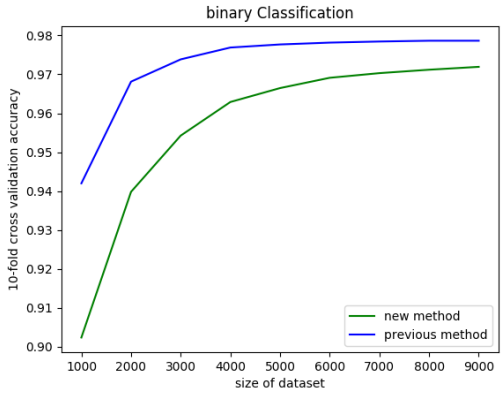

Figure 14. Size of dataset vs accuracy in binary classification with synthetic data

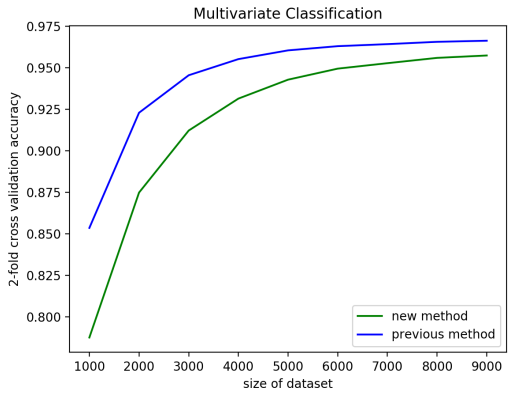

Figure 15. Size of dataset vs accuracy in multivariate classification with synthetic data

Experiment limitation:

- In both experiments done on the original dataset, the accuracy has not yet reach a plateau. Hypothetically, we we have more data, we should be able to reach a higher accuracy. This is shown by the experiments done on the synthetic dataset. However, since we are reusing every graph for 30 times, there might be overfitting in the synthetic dataset experiment.
- Node-one-hot encoding and edge-one-hot encoding relies on the number of nodes and edges contained in the data set. If it is given a graph that contains nodes or edges that is not in the data set where the estimator is trained, the estimator is not able to make a decision based on the unseen nodes/edges in the dataset.

Conclusion

- Node2vec-based distance featurization is able to integrate more information into the vector than the previous method. One must carefully choose the node2vec parameters that accurately reflects the topological relationships between nodes on the graph.
- Binary classification is more accurate compared to multivariate classification, since it has more samples per label. We believe the multi-variant classifier can do better with a bigger data set.
- The proposed new featurization approach does slightly worse than the previous method when run on a medium size real world dataset. There are several possible reasons: (1) The choice of **node2vec hyper-parameters** does not accurately reflects the topological features on the graph, (2) **The sample sizes** (less than 1K) are too small so that there aren't sufficiently strong signal for the classifier to build a good decision tree (3) **The sample labels might be inaccurate** according to our analysis and prior experience of using this dataset.
- For future work, we can experiment with more parameters in node2vec which may give better performance. We are mostly concerned with the parameters that affect the random walks, but the parameters that affect the word embedding for the random walk may also result in a positive influence of the featurization. We can also evaluate this approach on a different dataset.
- Moreover, convolutional layers and relevant deep learning techniques can be used to detect if a certain patterns of graph structure exists and the way different structures interact with each other.