Probabilistic Graph Model for Pure Data

1 Methodology of the Graph Probabilistic Model

Our goal is to output a probability value of a Pure Data (PD) program subgraph given a corpus of subgraphs extracted from the parsed PD files. Our code is available in GitHub [1]. The methodology is as follows:

Data preparation: We partition the PD projects into training and testing sets and extract parsed PD files for each dataset.

Graph Construction: We build a graph from the list of connections in a parsed PD file. After that, we extract the unique nodes such as `msg`, `tgl`; count the frequency of each node; and save them in our corpus.

Subgraph Analysis: For each unique connection or edge, we create adjacency matrices, calculate the frequencies of the unique connections, and save them in our corpus. In addition, we find the three-node subgraphs within the input graph, generate adjacency matrices for each, calculate the frequencies of the 3-node subgraphs, and store them in our corpus.

Research Questions: We address the following questions:

1. Given a collection of subgraphs along with their respective probabilities, what is the likelihood of a particular subgraph?
2. In the scenario of a subgraph featuring an empty node, what is the most probable node to occupy the `BLANK` position?

Evaluation: Finally, we evaluate the performance of our graph based model using our test dataset.

We explain the stages in the following sections.

1.1 Data Preparation

We partitioned the PD projects into an 80-20 split for training and testing purposes, utilizing the `train_test_split` [2] function from the `scikit-learn` [3] library in Python.

We gathered the hashes corresponding to the revisions of the PD file contents for both the train and test projects. Following this, we refined the test hashes by eliminating any matches found in the training set to assess our model's performances on paths generated from previously unseen parsed contents. This process guarantees that our training and test data originate from distinct projects, minimizing overlaps between them.

1.2 Graph Construction

We utilized the parsed contents of the PD files from our dataset [4] to generate graphs from the PD source code. We extracted a list of connections between the PD file objects using the `Contents` table from our dataset for each parsed content. We constructed a directed graph using Python's `NetworkX` [5] library for each parsed content. For instance, if there was a connection from `object_0` to `object_1`, we added both `object_0` and `object_1` as nodes in the graph, if not already added, and created a directed edge between them. In this context, `object_0` represents the `source` of the connection, while `object_1` is the `destination`.

Following that, we computed the frequency of each unique node within the graph and stored the results. Our training set comprises 34,566 unique nodes, with the most frequent ones being `msg`, occurring 2,880,151 times across all parsed PD programs in the training set; `r`, occurring 1,311,419 times; and `t`, appearing 1,195,537 times. Message boxes are containers for one or more messages, which are transmitted to their designated outlets or destinations upon activation of the box [6, 7]. Notably, `r` stands for `receive`, responsible for message reception, while `t` denotes `trigger`, facilitating message sequencing and conversion [6, 8, 7].

1.3 Subgraph Analysis

For each PD file graph, we extracted information about the 2-node and 3-node subgraphs to create our corpus. The 2-node subgraphs can be extracted by investigating the connections or edges in the graph. We also generated paths with length 2 for each node using the `all_simple_paths` [9] function from NetworkX. We extracted the 3-node subgraphs by extracting paths of length 2 and consisting of 3 nodes from the graph.

We collected data on 2-node and 3-node subgraphs from each PD file graph to build our training subgraph corpus. For the 2-node subgraphs, we examined the graph's edges to understand their connectivity. We then employed the `all_simple_paths` [9] function from NetworkX to create paths of length 2 from every single node in the undirected version of the same graph. In the case of 3-node subgraphs, our approach was to pinpoint all 2-length paths that included 3 nodes, thereby mapping the graph's subgraphs consisting of 3 nodes.

For each of the connected 2-nodes and 3-nodes, we arrange the nodes based on their object type (e.g., `msg`, `tgl`), then construct the adjacency matrix of the subgraph. In this matrix, the first row represents connections originating from the first sorted node, the second row represents connections from the second sorted node, and so forth. We have 217,806 unique 2-node subgraphs in our training corpus.

Subsequently, we generate a vector of length $|unique\ nodes| + 1$, where the last entry is reserved for unseen nodes during testing. Each element of this vector corresponds to a word from our unique node list, with entries set to 0 except for positions corresponding to nodes present in the 3-node subgraph. In such cases, the entry equals the frequency of that particular type of node in the subgraph. For example, suppose our unique node list includes `[floatatom, msg, r, s, t, tgl]`, and the 3-node subgraph comprises the following node types: `[floatatom, msg, msg]`. In this case, the word vector representing this specific subgraph would be `[1,` `2, 0, 0, 0, 0, 0]`. We have 1,285,331 unique 3-node subgraphs in our training corpus.

Finally, we combine the word vector and adjacency matrix, creating a sha256 [10] key to uniquely identify the subgraph. We record the frequency of each subgraph to facilitate probability calculations.

1.4 Research Questions

After constructing our corpus of subgraphs, we try to answer two research questions:

RQ1: Given a collection of subgraphs along with their respective probabilities, what is the likelihood of a particular subgraph?

RQ2: In the scenario of a subgraph featuring an empty node, what is the most probable node to occupy the `BLANK` position?

1.4.1 RQ1: Calculating the probability of a given subgraph based on our corpus of subgraphs

Initially, for a given 3-node subgraph, we compute its identifying key using the procedure outlined in Section 1.3. Subsequently, we determine the probability of a subgraph using the following algorithm:

1. If the key is present in our corpus, the score is assigned as the probability of the 3-node subgraph. This probability is calculated as the frequency of the 3-node subgraph divided by the sum of the frequencies of all 3-node subgraphs.

2. In cases where the key is absent from our corpus, we generate keys for the 2-node subgraphs using the same procedure (refer to Section 1.3). For each 2-node subgraph in our input graph, if its key exists in the corpus, we multiply the final probability score by the probability of the 2-node subgraph. The probability of a 2-node subgraph is calculated as the frequency of the current 2-node subgraph divided by the sum of the frequencies of all 2-node subgraphs in our corpus.

3. If the 2-node subgraphs are not present in our corpus, we adjust the score by the probability of the nodes within the subgraph. The probability of a 1-node subgraph is determined by the frequency of that node divided by the sum of the frequencies of all unique nodes. Additionally, if the node is not present in our corpus, the score is multiplied by a very small value, such as $0.5 \times$ (1/sum of the frequencies of all unique nodes).

4. For each missed 3-node and 2-node subgraph, the score is multiplied by a small discount factor to indicate that these subgraphs are unseen in our corpus, thus their probabilities should be lower compared to those observed in our corpus. We have used a discount factor of 0.05.

Finally, we return the score computed through the aforementioned algorithm to estimate the likelihood of a subgraph.

1.4.2 Predicting the empty node of a given subgraph

To predict the node in the empty position of the input subgraph, we follow the methodology outlined in Section 1.4.1. However, to identify the node occupying the empty position, we iterate through our entire vocabulary and select the type of node that maximizes the subgraph's score.

For an input 3-node subgraph, we randomly remove one node and substitute it with each node from our vocabulary while preserving the edges. This process enables us to determine which substitution generates the highest score. Subsequently, we return the corresponding node from our vocabulary as the predicted node.

1.5 Evaluation

For every parsed content extracted from our test hashes, we construct 3-node subgraphs. For each subgraph, we employ the methodology outlined in Section 1.4.2 to predict the node for the empty position. Subsequently, we compute accuracy using the following formula:

accuracy = number of correct predictions/number of total predictions

2 Limitations

There are some limitations to our model which are listed below:

2.0.1 Corpus limited to 3 and 2 node subgraphs

Our corpus is constructed solely from 2-node and 3-node subgraphs. Furthermore, during evaluation, we restricted ourselves to utilizing 3-node subgraphs. However, the model's performance might have seen improvement had we incorporated larger subgraphs since doing so would have provided additional contexts for predicting unknown tokens.

References

[1] Anisha Islam. Probability Estimator for Visual Code. `https://github.com/anishaislam8/Probability-Estimator-For-Visual-Code`. Accessed: 2024-05-14.

[2] scikit-learn developers (BSD License). sklearn.model_selection.train_test_split. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`. Accessed: 2024-02-19.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] Anisha Islam. Opening the Valve on Pure-Data Dataset. `https://archive.org/details/Opening_the_Valve_on_Pure_Data`, 2023. Accessed: 2024-02-19.

[5] NetworkX Developers. NetworkX. https://networkx.org/. Accessed: 2024-02-19.

[6] Miller Puckette et al. Pure Data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.

[7] IEM. Pure Data. https://puredata.info/. Accessed: 2024-01-20.

[8] K. Barkati and N. Granieri. Pure Data Reference Card. https://puredata.info/docs/manuals/pdrefcards/pd-refcard-en.pdf/view, 2018. Accessed: 2023-11-12.

[9] NetworkX Developers. all_simple_paths. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.simple_paths.all_simple_paths.html. Accessed: 2024-02-19.

[10] FIPS Pub. Secure Hash Standard (SHS). *Fips pub*, 180(4), 2012.