Graph Neural Networks and Code Completion

Kalyan N. (IMT2018034) Naveen Kumar V R(IMT2017029)

1 Introduction

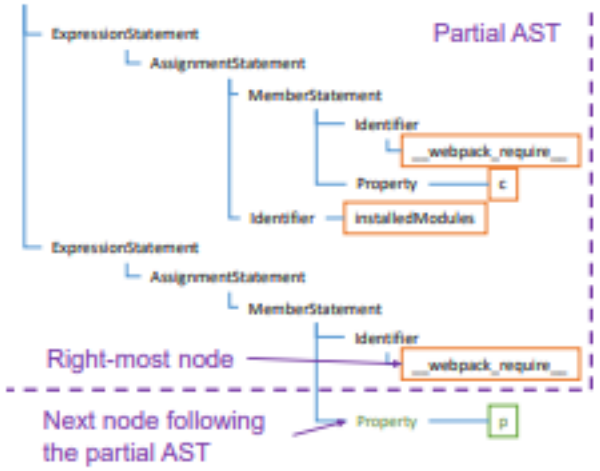
Intelligent code completion is a fundamental component of all modern Integrated Development Environment (IDEs) and is the first step towards complete code generation. Most intelligent code completion approaches rely on strong typing and other relevant compiler information. While this is an effective method for dealing with statically typed languages like C++ and Java, it limits their applicability to other popular languages which are dynamically typed languages like JavaScript and Python.

Neural methods have shown great promise in tackling this problem, a key breakthrough was the utilization of the Abstract Syntax Tree (AST) of the program for code completion. In this project we hope to extend prior work in the field and improve accuracy of code completion by explicitly exploiting the structure of Abstract Syntax Trees using Graph Neural Networks (GNNs).

2 Related Work

In the past the approaches that relied on building probabilistic models can be categorized as n-gram models, probabilistic grammars and log-bilinear models. In the first attempt of using neural methods for code completion they explored the application of Recurrent Neural Networks (RNNs) for code completion. In 2017, Liu et al. extended this work by taking the pre-order traversal of the AST and feeding the tokens into an Long Short Term Memory (LSTM) and attempting to predict the next token of the sequence which corresponds to the next node in the AST. Following this, more powerful deep learning techniques including the use of transformers and its variants were used for code completion [3, 2].

3 Problem Statement



Given a partial AST the task is to predict the next node following it in the partial AST. Nodes can be classified as either terminals or non-terminals. Terminals are leaves of the trees which include constants, function

names, identifiers, etc. While non-terminals are the intermediary nodes which include loops, conditionals, expressions, etc. The non-terminals define the structure of the program while the terminals are the exact values. The prediction of the next node can be classified into two problems: the prediction of the next terminal and the prediction of the next non-terminal. Note that each non-terminal has at most one terminal. We restrict the scope of our project to predicting the next non-terminal.

4 Dataset

The dataset we used was a compilation of 150,000 JavaScript files scraped from GitHub and their corresponding ASTs. This dataset was released by the Software Reliability Lab at ETH Zurich.

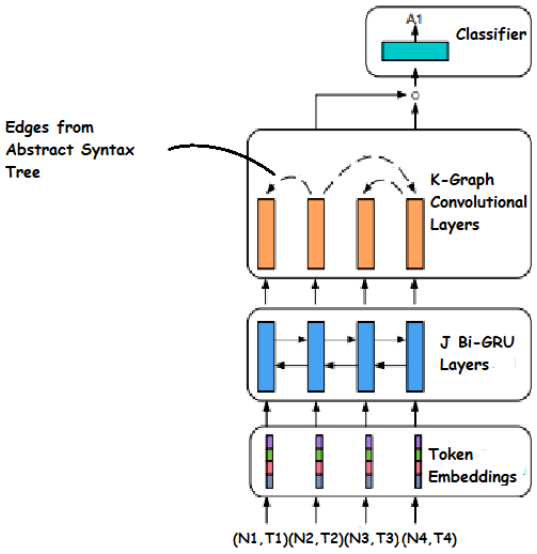
5 Pre-Processing

Given a partial AST as input we serialize it according to it's depth first search (DFS) traversal and construct a sequence (N_1, T_1), $(N_2, T_2) \cdots (N_k, T_k)$ and attempt to predict the next token in the sequence. For every non-terminal there is at most one corresponding terminal which is the value T_i for a given N_i. Many non-terminals don't have a corresponding terminal so we introduce an artificial dummy token called 'none' for all of these non-terminals. We retrieve the top 10,000 occurring terminals in the train dataset and mark the remaining as UNK. We also add an End of File (EOF) token at the end of each program.

6 Model Used

Graph Neural Networks have shown impressive results in various areas of NLP and are the state of the art for many open problems. Our work is inspired by the success of the usage of Graph Convolutional Networks (GCNs) on syntax trees to enrich the embeddings produced by the LSTM. [4]

Prior work has shown that LSTMs and GCNs are complementary in nature. While LSTMS are able to capture contextual embeddings, GCNs over syntax trees help learn syntactic information which LSTM's struggle with. [5] We explore a similar combination as well.



The topology of our model is depicted above.

Given an input sequence of tokens, the embedding of each token is computed as:

$$E_i = AN_i \circ BT_i$$

Where $\circ$ is the concatenation operation, A is a $J \times V_N$ matrix and B is a $K \times V_i$ matrix, where V_N and V_i are the vocabulary sizes of non-terminal and terminal respectively. J is the embedding size for non terminal and K is the embedding size for terminals.

These embeddings are then fed into a bi-directional Gated Recurrent Unit (GRU) with J layers. The output at each time step/token is the matrix h_i where h_i belongs to $\mathbb{R}^F$ where F is hidden layer dimension.

These embeddings are then passed into graph convolutional layers. For our experiments we use Graph Attention Layers [**?**].

Graph attention is similar to 'regular' attention [6], it computes an attention matrix which captures how important a node i's features are for node j.

It takes in set of note/token features $h = \{\vec{h_1}, \vec{h_2}, \cdots, \vec{h_N}\}$, $\vec{h_i} \in \mathbb{R}^F$ and outputs another set of node/token features $h' = \{\vec{h'_1}, \vec{h'_2}, \cdots, \vec{h'_N}\}$, $\vec{h'_i} \in \mathbb{R}^{F'}$

It calculates the attention coefficient e_{ij} by the below formulation:

$$e_{ij} = a(W\vec{k_i}, W\vec{k_j})$$

W is a linear transformation, $W \in \mathbb{R}^{F' \times F}$ and $a : \mathbb{R}^F \times \mathbb{R}^{F'} \to \mathbb{R}$ computes attention coefficients.

The attention coefficients are then normalized across J using the softmax function.

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum\limits_{k \in \mathbb{N}_i} \exp(e_{ik})}$$

Finally, h'_i is obtained by taking a linear combination of all neighbours of i, weighted by attention coefficient

$$h'_i = \sum_{i \in N_i} \alpha_{ij} W\vec{h_i}$$

This mechanism can also be extended to multihead attention similar to [7] but for our experiment we use a single head of attention.

We take h'_N, apply a linear transformation w_L, where $W_L \in \mathbb{R}^{F' \times V_N}$, which transforms the hidden vector to another dimension $\mathbb{R}^{V_N}$, where V_N is the vocabulary of non-terminals.

This is passed through a softmax layer and trained with cross-entropy loss.

7 Experiments and Training Details

We trained all our models on only a subset of the training data due to the large dataset size and hence training time. We split programs into sequences of 50 tokens each and the last sequence of a program was padded with EOF characters to ensure it was 50 tokens long. We experimented with batch sizes 64, 128 and 256, 128 provided quickest convergence. When we pass in a sequence of a program we retrieve hn and use this as the initial hidden state for the next sequence of the program. This limited our ability to parallelize, and forced very long training times.

We experiment with 2 models, the first being a GRU based model. Both the models were trained using the Pytorch library. For the second model where we included graph attention layers, Pytorch Geometric was used as well.

The architecture is shown below, it has embedding layers one for non-terminals and one for terminals. After which it is passed into 2 layers of bi-directional GRU's, upon which a linear layer is applied and we take the softmax.

```
Graph2CodeNet(
  (embeddingLayervalue): Embedding(10002, 128)
  (embeddingLayertype): Embedding(185, 64)
  (GRU): GRU(192, 512, num_layers=2, batch_first=True, bidirectional=True)
  (linearTerminal): Linear(in_features=2048, out_features=10002, bias=True)
  (linearNon_Terminal): Linear(in_features=2048, out_features=46, bias=True)
)
```

We trained this model on 40,000 files, with a LR of 0.01. We employ gradient clipping strategy to prevent a common issue in RNN based models which is vanishing/exploding gradients [8].

On a completely unseen subset of the dataset (not used for tuning hyperparameters or training) we obtain an accuracy of 72.52%. The training took 2 hours to pass through the 40,000 files once.

```
Graph2CodeNet(
  (embeddingLayervalue): Embedding(10002, 128)
  (embeddingLayertype): Embedding(185, 64)
  (GRU): GRU(192, 512, num_layers=2, batch_first=True, bidirectional=True)
  (att1): GATConv(1024, 2048, heads=1)
  (att2): GATConv(2048, 2048, heads=1)
  (linearNon_Terminal): Linear(in_features=2048, out_features=46, bias=True)
)
```

The next model we used below was the same with the addition of 2 graph attention layers. A significant issue we faced with this model was the large size of the edge lists of the corresponding ASTs. This proved to be a bottleneck for the number of programs we could store in the RAM. Just 7000 programs were able to fill up a RAM size of 32gb. For this reason we employed a generator based approach, each file was parsed and then ran before the next file was parsed. This slowed down training even further. It took 8 hours to run through 40,000 files just once. 16 hours to run through 80,000 files once which was what we trained our model finally on. This model implemented gradient clipping as well. On a completely unseen subset of the dataset we obtained an accuracy of 74.5%. Due to the extremely large training time we were unable to tune hyperparameters or try various adjustments. We believe that GNNs show a lot more promise but we were unable to try the variations we wished to or even train on the complete dataset till validation score peaks. Some further explorations/experiments we would have liked to run on our GNN model are:

- Using different edge types for the edges of the AST(forward and backward). An AST is a Directed Acyclic Graph(DAG). We believe message propagation should be bi-directional and have implemented it so. But the way the model treats the edges can be different. Similarly we can have an additional edge type for self loops.
- Varying training hyperparameters and also additionally hyperparameters of the model, like attention heads, different convolutional layers, etc.
- Pass node number of the sequence as a node feature. Our belief is that the model can learn how to weigh edges appropriately by how large the distance b/w the 2 nodes is.
- By splitting our programs into sequences of size 50. We lose the ability to have edges from one segment of the program to another. If we work around this, these edges have the ability to provide long term syntactic information, which an LSTM can't handle.

8 Conclusion

GNNs are a very powerful new learning paradigm especially due to the omnipresence of graphs in the real world. This was a first step in exploring the applications of GNNs for code completion and we believe this avenue holds a lot of promise.

References

[1] Liu, C., Wang, X., Shin, R., Gonzalez, J. E., and Song, D., (2016) *Neural Code Completion.* Available at: `https://openreview.net/pdf?id=rJbPBt9lg` (Accessed: 19.05.2021)

[2] Liu, F., Li, G., Wei, B., Xia, X., Fu, Z., and Jin, Z. (2020) *A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning.* New York: ACM.

[3] Kim, S., Zhao, J., Tian, Y., and Chandra, S., (2020) *Code Prediction by Feeding Trees to Transformers.* Available at: `https://arxiv.org/abs/2003.13848` (Accessed: 19.05.2021)

[4] Marcheggiani, D., and Titov, I. (2017) *Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling.* Copenhagen, Denmark: Association for Computational Linguistics.

[5] Bastings, J., Titov, I. Aziz, W., Marcheggiani, D., and Sima'an, k. (2017) *Graph Convolutional Encoders for Syntax aAare Neural Machine-Translation.* Copenhagen, Denmark: Association for Computational Linguistics.

[6] Bahdanau, D., Cho, K., and Bengio, Y. (2015) *Neural Machine Translation By Jointly Learning to Align and Translate.* Available at: `https://arxiv.org/pdf/1409.0473.pdf`. (Accessed: 19.05.2021)

[7] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A N., Kaiser, L., and Polosukhin, I. (2017) *Attention Is All You Need.* New York: Curran Associates Inc.

[8] Pascanu, R., Mikolov, T., and Bengio, Y. (2013) *On the difficulty of training recurrent neural networks.* Available at: `http://icml.cc/2013/` (Accessed: 19.05.2021)