Neural Program Synthesis from Diverse and Distant Context

Xinyun Chen, Linyuan Gong, Alvin Cheung, Dawn Song

1 Overview

There has been an increasing interest of studying deep learning techniques to synthesize programs from specifications, including natural language descriptions, input-output examples, code contexts, etc. Despite that some existing work demonstrates promising results under the controlled setting, e.g., the specifications are short and standardized, the model performance significantly degrades when the specifications become complex and include multiple components. For example, Agashe et al. collected JuICe, a large-scale dataset with Python Jupyter notebooks scrawled from GitHub, and proposed a task that synthesizes the code from the natural language markdown and code cells above. However, the evaluation shows that existing neural sequence generation models, including LSTMs and Transformers, have a hard time generating reasonable code to be useful. Therefore, how to properly leverage information from diverse and distant contexts for neural program synthesis remains a grand challenge.

In this project, we aim to synthesize visualization programs using a combination of natural language utterances and the programmatic context that the visualization program will reside (e.g., code written in the same file as the visualization program to load the plotted data), focusing on programs that create static visualizations (e.g., line charts, scatter plots, etc). We design a *hierarchical* deep neural network code generation model called PLOTCODER that decomposes synthesis into two subtasks: generating the plot command, then the parameters to pass in given the command. PLOTCODER uses a pointer network architecture, which allows the model to directly select code tokens in the previous code cells in the same notebook as the plotted data. Meanwhile, inspired by the schema linking techniques proposed for semantic parsing with structured inputs, such as text to SQL tasks, PLOTCODER's encoder connects the embedding of the natural language descriptions with their corresponding code fragments in previous code cells within each notebook. Although the constructed links can be noisy because the code context is less structured than the database tables in text-to-SQL problems, we observe that our approach results in substantial performance gain.

We evaluate PLOTCODER's ability to synthesize visualization programs using Jupyter notebooks of homework assignments or exam solutions. On the gold test set where the notebooks are official solutions, our best model correctly predicts the plot types for over 80% of samples, and precisely predicts both the plot types and the plotted data for over 50% of the samples. On the more noisy test splits with notebooks written by students, which may include work-in-progress code, our model still achieves over 70% plot type prediction accuracy, and around 35% accuracy for generating the entire code, showing how PLOTCODER's design decisions improve our prediction accuracy.

Our paper is accepted in ACL 2021: `https://aclanthology.org/2021.acl-long.169/`. We present the key results below, and more details can be found in the paper.

2 PlotCoder Architecture

As shown in Figure 1, PLOTCODER includes an LSTM-based encoder to jointly embed the natural language and code context, as well as a hierarchical decoder that generates API calls and selects data for plotting.

2.1 NL-Code Context Encoder

PLOTCODER's encoder computes a vector representation for each token in the natural language description and the code context. For each token in the code context that also occurs in the natural language, we denote h_c and h_{nl} as the embedding vectors computed by the code context encoder LSTM_c and the NL encoder LSTM_{nl}.

NL-code linking. Capturing the correspondence between the code context and natural language is crucial in achieving a good prediction performance. Therefore, we design the NL-code linking mechanism to explicitly connect the embedding vectors of code tokens and their corresponding natural language words. Specifically, we compute a new code token embedding vector as:

$$h'_c = W_l([h_c; h_{nl}])$$

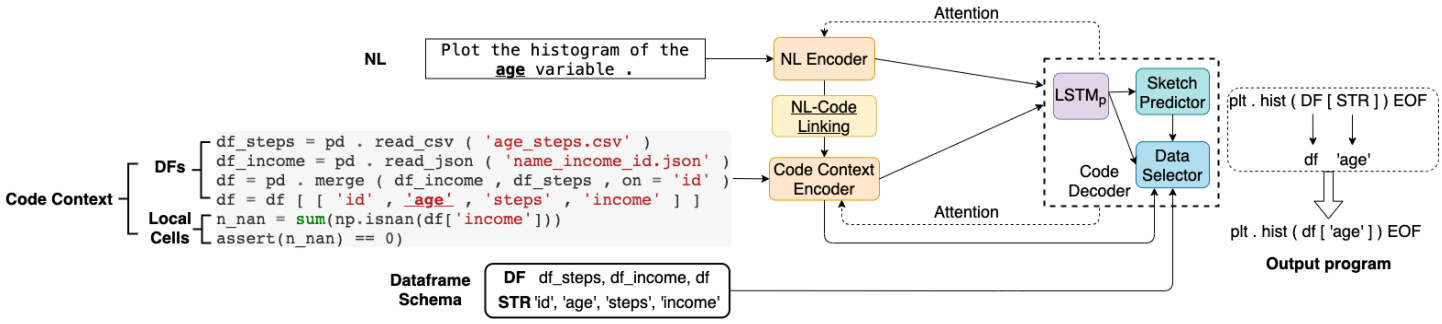

Figure 1: *Overview of the* PLOTCODER *architecture. The NL-Code linking component connects the embedding vectors for underscored tokens in natural language and code context, i.e., "age".*

where W_l is a linear layer, and $[h_c; h_{nl}]$ is the concatenation of h_c and h_{nl}.

2.2 Hierarchical Program Decoder

We train another LSTM to decode the visualization code sequence, denoted as LSTM$_p$. Our decoder generates the program in a hierarchical way. At each timestep, the model first predicts a token from the code token vocabulary that represents the program sketch. As shown in Figure 1, the program sketch does not include the plotted data. After that, the decoder predicts the plotted data, where it employs a copy mechanism to select tokens from the code context.

3 Results

Table 1: Evaluation on program accuracy.

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
Full Model	**34.79%**	**34.70%**	**56.25%**	**47.37%**
− Hierarchy	30.20%	31.56%	45.83%	**47.37%**
− Link	29.98%	28.05%	43.75%	45.61%
LSTM	26.17%	24.67%	41.67%	40.35%
+ CodeBERT	33.11%	34.58%	54.17%	35.09%
+ RoBERTa	32.77%	33.37%	50.00%	26.32%

Table 2: Evaluation on plot type accuracy.

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
Full Model	**70.58%**	**71.46%**	83.33%	78.95%
− Hierarchy	64.65%	68.92%	**87.50%**	82.46%
− Link	65.32%	64.09%	81.25%	73.68%
LSTM	66.67%	67.47%	85.42%	**85.96%**
+ codeBERT	65.44%	67.96%	75.00%	57.89%
+ RoBERTa	65.21%	66.38%	66.67%	54.39%

We present the program accuracy and plot type accuracy results in Table 1 and Table 2 respectively. On the hard data splits, the hierarchical PLOTCODER predicts 35% of the samples correctly, improving over the non-hierarchical model by $3-4.5\%$. Meanwhile, NL-code linking enables the model to better capture the correspondence between the code context and the natural language, and consistently improves the performance. Without the copy mechanism, the baseline LSTM cannot predict any token outside of the code vocabulary. Therefore, this model performs worse than other LSTM-based models. Interestingly, using BERT-like encoders does not improve the results. This might be due to the difference in data distribution for pre-training and vocabularies.