PLOTCODER: Hierarchical Decoding for Synthesizing Visualization Code in Programmatic Context

Xinyun Chen, Linyuan Gong, Alvin Cheung and Dawn Song
UC Berkeley
{xinyun.chen, gly, akcheung, dawnsong}@berkeley.edu

Abstract

Creating effective visualization is an important part of data analytics. While there are many libraries for creating visualizations, writing such code remains difficult given the myriad of parameters that users need to provide. In this paper, we propose the new task of synthesizing visualization programs from a combination of natural language utterances and code context. To tackle the learning problem, we introduce PLOTCODER, a new hierarchical encoder-decoder architecture that models both the code context and the input utterance. We use PLOTCODER to first determine the template of the visualization code, followed by predicting the data to be plotted. We use Jupyter notebooks containing visualization programs crawled from GitHub to train PLOTCODER. On a comprehensive set of test samples from those notebooks, we show that PLOTCODER correctly predicts the plot type of about 70% samples, and synthesizes the correct programs for 35% samples, performing 3-4.5% better than the baselines.[1]

1 Introduction

Visualizations play a crucial role in obtaining insights from data. While a number of libraries (Hunter, 2007; Seaborn, 2020; Bostock et al., 2011) have been developed for creating visualizations that range from simple scatter plots to complex 3D bar charts, writing visualization code remains a difficult task. For instance, drawing a scatter plot using the Python matplotlib library can be done using both the `scatter` and `plot` methods, and the `scatter` method (Matplotlib, 2020) takes in 2 required parameters (the values to plot) along with 11 other optional parameters (marker type, color, etc), with some parameters having numeric types (e.g., the size of each marker) and some being arrays (e.g., the list of colors for each collection of the plotted data, where each color is specified as a string or another array of RGB values). Looking up each parameter's meaning and its valid values remains tedious and error-prone, and the multitude of libraries available further compounds the difficulty for developers to create effective visualizations.

In this paper, we propose to *automatically 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). While there has been prior work on synthesizing code from natural language (Zettlemoyer and Collins, 2012; Oda et al., 2015; Wang et al., 2015; Yin et al., 2018), and with addition information such as database schemas (Zhong et al., 2017; Yu et al., 2018, 2019b,a) or input-output examples (Polosukhin and Skidanov, 2018; Zavershynskyi et al., 2018), synthesizing general-purpose code from natural language remains highly difficult due to the ambiguity in the natural language input and complexity of the target. Our key insight in synthesizing visualization programs is to leverage their properties: they tend to be short, do not use complex programmatic control structures (typically a few lines of method calls without any control flow or loop constructs), with each method call restricted to a single plotting command (e.g., `scatter`, `pie`) along with its parameters (e.g., the plotted data). This influences our model architecture design as we will explain.

To study the visualization code synthesis problem, we use the Python Jupyter notebooks from the JuiCe dataset (Agashe et al., 2019), where each notebook contains the visualization program and its programmatic context. These notebooks

[1] Our code and data are available at https://github.com/jungyhuk/plotcoder.

Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pages 2169–2181
August 1–6, 2021.

are crawled from GitHub and written by various programmers, thus a main challenge is understanding the complexity and the noisiness of real-world programmatic contexts and the huge variance in the quality of natural language comments. Unfortunately, using standard LSTM-based models and Transformer architectures (Vaswani et al., 2017) fails to solve the task, as noted in prior work (Agashe et al., 2019).

We observe that while data to be plotted is usually stored in pandas dataframes (Pandas, 2020), they are not explicitly annotated in JuiCe. Hence, unlike prior work, we augment the programmatic context with dataframe names and their schema when available in predicting the plotted data.

We next utilize our insight above and 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 (Vinyals et al., 2015), 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 (Iyer et al., 2017; Wang et al., 2019a; Guo et al., 2019), 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.

Natural Language

Explore the relationship between rarity and a skill of your choice. Choose one skill ('Attack','Defense' or 'Speed') and do the following. Use the scipy package to assess whether `Catch_Rate` predicts the skill. Create a scatterplot to visualize how the skill depends upon the rarity of the pokemon. Overlay a best fit line onto the scatterplot.

Local Code Context

```
slope, intercept, r_value, p_value, std_err =
   linregress(df['Catch_Rate'], df['Speed'],)
x = np.arange(256)
y = slope * x + intercept
```

Distant Dataframe Context

```
df['Weight_kg'].describe()
df['Color'].value_counts().plot(kind='bar')
df['Body_Style'].value_counts().plot(kind='bar')
grouped = df.groupby(['Body_Style','hasGender',]).mean()
df.groupby('Color')['Attack'].mean()
df.groupby('Color')['Pr_Male'].mean()
df.sort_values('Catch_Rate',ascending=False).head()
```

Dataframe Schema

```
df: ['Catch_Rate', 'Speed', 'Weight_kg', 'Color',
     'Body_Style']
```

Ground Truth

```
plt.scatter(df['Catch_Rate'], df['Speed'])
plt.plot(x,y)
```

Figure 1: An example of plot code synthesis problem studied in this work. Given the natural language, code context within a few code cells from the target code, and other code snippets related to dataframes, PLOTCODER synthesizes the data visualization code.

2 Related Work

There has been work on translating natural language to code in different languages (Zettlemoyer and Collins, 2012; Wang et al., 2015; Oda et al., 2015; Yin et al., 2018; Zhong et al., 2017; Yu et al., 2018; Lin et al., 2018). While the input specification only includes the natural language for most tasks, prior work also uses additional information for program prediction, including database schemas and contents for SQL query synthesis (Zhong et al., 2017; Yu et al., 2018, 2019b,a), input-output examples (Polosukhin and Skidanov, 2018; Zavershynskyi et al., 2018), and code context (Iyer et al., 2018; Agashe et al., 2019). There has also been work on synthesizing data manipulation programs only from input-output examples (Drosos et al., 2020; Wang et al., 2017). In this work, we focus on synthesizing visualization code from both natural language description and code context, and we construct our benchmark based on the Python Jupyter notebooks from the JuiCe (Agashe et al., 2019). Compared to JuiCe's input format, we also annotate dataframe schema if available, which is especially important for visualization code synthesis.

Prior work has studied generating plots from other specifications. Falx (Wang et al., 2019b,

2021) synthesizes plots from input-output examples, but do not use any learning technique, and focuses on developing a domain-specific language for plot generation instead. In (Dibia and Demiralp, 2019), the authors apply a standard LSTM-based sequence-to-sequence model with attention for plot generation, but the model takes in only raw data to be visualized with no natural language input. The visualization code synthesis problem studied in our work is much more complex, where *both* the natural language and the code context can be long, and program specifications are implicit and ambiguous.

Our design of hierarchical program decoding is inspired by prior work on sketch learning for program synthesis, where various sketch representations have been proposed for different applications (Solar-Lezama, 2008; Murali et al., 2018; Dong and Lapata, 2018; Nye et al., 2019). Compared to other code synthesis tasks, a key difference is that our sketch representation distinguishes between dataframes and other variables, which is important for synthesizing visualization code.

Our code synthesis problem is also related to code completion, i.e., autocompleting the program given the code context (Raychev et al., 2014; Li et al., 2018; Svyatkovskiy et al., 2020). However, standard code completion only requires the model to generate a few tokens following the code context, rather than entire statements. In contrast, our task requires the model to synthesize complete and executable visualization code. Furthermore, unlike standard code completion, our model synthesizes code from both the natural language description and code context. Nevertheless, when the prefix of the visualization code is given, our model could also be used for code completion, by including the given partial code as part of the code context.

3 Visualization Code Synthesis Problem

We now discuss our problem setup of synthesizing visualization code in programmatic context, where the model input includes different types of specifications. We first describe the model inputs, then introduce our code canonicalization process to make it easier to train our models and evaluate the accuracy, and finally our evaluation metrics.

3.1 Program Specification

We illustrate our program specification in Figure 1, which represents a Jupyter notebook fragment. Our task is to synthesize the visualization code given the natural language description and code from the preceding cells. To do so, our model takes in the following inputs:

- The natural language description for the visualization, which we extract from the natural language markdown above the target code cell containing the gold program in the notebook.
- The local code context, defined as a few code cells that immediately precede the target code cell. The number of cells to include is a tunable hyper-parameter to be described in Section 5.
- The code snippets related to dataframe manipulation that appear before the target code cell in the notebook, but are not included in the local code context. We refer to such code as the distant dataframe context. When such context contains code that uses dataframes, they are part of the model input by default.

As mentioned in Section 1, unlike JuiCe, we also extract the code snippets related to dataframes, and annotate the dataframe schemas according to their syntax trees. As shown in Figure 1, knowing the column names in each dataframe is important for our task, as dataframes are often used for plotting.

3.2 Code Canonicalization

One way to train our models is to directly utilize the plotting code in Jupyter notebooks as the ground truth. However, due to the variety of plotting APIs and coding styles, such a model rarely predicts exactly the same code as written in Jupyter notebooks. For example, there are at least four ways in Matplotlib (and similar in other libraries) to create a scatter plot for columns `'y'` against `'x'` from a dataframe `df`: `plt.scatter(df['x'],df['y'])`, `plt.plot(df['x'],df['y'],'o')`, `df.plot.scatter(x='x',y='y')`, `df.plot(kind='scatter',x='x',y='y')`. Moreover, given that the natural language description is ambiguous, many plot attributes are hard to precisely predict. For example, from the context shown in Figure 1, there are many valid ways to specify the plot title, the marker style, axis ranges, etc. In our experiments, we find that when trained on raw target programs, fewer than 5% predictions are exactly the same as the ground truth, and a similar phenomenon is also observed earlier (Agashe et al., 2019).

Therefore, we design a canonical representation for plotting programs, which covers the core of plot generation. Specifically, we convert the plotting

code into one of the following templates:

- `LIB.PLOT_TYPE(X,{Y}*)`, where `LIB` is a plotting library, and `PLOT_TYPE` is the plot type to be created. The number of arguments may vary for different `PLOT_TYPE`, e.g., 1 for histograms and pie charts, and 2 for scatter plots.
- L_0 `\n` L_1 `\n` ... L_m, where each L_i is a plotting command in the above template, and `\n` are separators.

For example, when using `plt` as the library (a commonly used abbreviation of `matplotlib.pyplot`), we convert `df.plot(kind='scatter',x='x',y='y')` into `plt.scatter(df['x'],df['y'])`, where `LIB = plt` and `PLOT_TYPE = scatter`. Plotting code in other libraries could be converted similarly.

The tokens that represent the plotted data, i.e., `X` and `Y`, are annotated in the code context as follows:

- `VAR`, when the token is a variable name, e.g., `x` and `y` in Figure 1.
- `DF`, when the token is a Pandas dataframe or a Python dictionary, e.g., `df` in Figure 1.
- `STR`, when the token is a column name of a dataframe, or a key name of a Python dictionary, such as `'Catch_Rate'` and `'Speed'` in Figure 1.

The above annotations are used to cover different types of data references. For example, a column in a dataframe is usually referred to as `DF[STR]`, and sometimes as `DF[VAR]` where `VAR` is a string. In Section 4.2, we will show how to utilize these annotations for hierarchical program decoding, where our decoder first generates a program sketch that predicts these token types without the plotted data, then predicts the actual plotted data subsequently.

3.3 Evaluation Metrics

Plot type accuracy. To compute this metric, we categorize all plots into several types, and a prediction is correct when it belongs to the same type as the ground truth. In particular, we consider the following categories: (1) scatter plots (e.g., generated by `plt.scatter`); (2) histograms (e.g., generated by `plt.hist`); (3) pie charts (e.g., generated by `plt.pie`); (4) a scatterplot overlaid by a line (e.g., such as that shown in Figure 1, or generated by `sns.lmplot`); (5) a plot including a kernel density estimate (e.g., plots generated by `sns.distplot` or `sns.kdeplot`); and (6) others, which are mostly plots generated by `plt.plot`.

Plotted data accuracy. This metric measures whether the predicted program selects the same data to plot as the ground truth. Unless otherwise specified, the ordering of variables must match the ground truth as well, i.e., swapping the data used to plot x and y axes result in different plots.

Program accuracy. We consider a predicted program to be correct if both the plot type and plotted data are correct. As discussed in Section 3.2, we do not evaluate the correctness of other plot attributes because they are mostly unspecified.

4 PLOTCODER Model Architecture

In this section, we present PLOTCODER, a hierarchical model architecture for synthesizing visualization code from natural language and code context. PLOTCODER includes an LSTM-based encoder (Hochreiter and Schmidhuber, 1997) to jointly embed the natural language and code context, as well as a hierarchical decoder that generates API calls and selects data for plotting. We provide an overview of our model architecture in Figure 2.

4.1 NL-Code Context Encoder

PLOTCODER's encoder computes a vector representation for each token in the natural language description and the code context, where the code context is the concatenation of the code snippets describing dataframe schemas and the local code cells, as described in Section 3.1.

NL encoder. We build a vocabulary for the natural language tokens, and train an embedding matrix for it. Afterwards, we use a bi-directional LSTM to encode the input natural language sequence (denoted as LSTM_{nl}), and use the LSTM's output at each timestep as the contextual embedding vector for each token.

Code context encoder. We build a vocabulary V_c for the code context, and train an embedding matrix for it. V_c also includes the special tokens {`VAR`, `DF`, `STR`} used for sketch decoding in Section 4.2. We train another bi-directional LSTM (LSTM_c), which computes a contextual embedding vector for each token in a similar way to the natural language encoder. We denote the hidden state of LSTM_c at the last timestep as H_c.

NL-code linking. Capturing the correspondence between the code context and natural language is crucial in achieving a good prediction performance. For example, in Figure 2, PLOTCODER infers that the dataframe column "age" should be plotted, as this column name is mentioned in the natural language description. Inspired by this observation, we

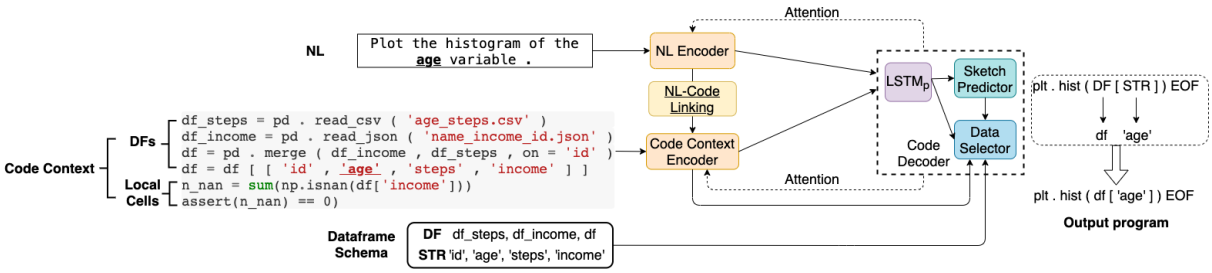

Figure 2: Overview of the PLOTCODER architecture. The NL-Code linking component connects the embedding vectors for underscored tokens in natural language and code context, i.e., "age".

design the NL-code linking mechanism to explicitly connect the embedding vectors of code tokens and their corresponding natural language words. Specifically, for each token in the code context that also occurs in the natural language, let h_c and h_{nl} be its embedding vectors computed by LSTM$_c$ and LSTM$_{nl}$, respectively, we compute a new code token embedding vector as:

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

where W_l is a linear layer, and $[h_c; h_{nl}]$ is the concatenation of h_c and h_{nl}. When no natural language word matches the code token, h_{nl} is the embedding vector of the `[EOS]` token at the end of the natural language description. When we include this NL-code linking component in the model, h'_c replaces the original embedding h_c for each token in the code context, and the new embedding is used for decoding. We observe that many informative natural language descriptions explicitly state the variable names and dataframe columns for plotting, which makes our NL-code linking effective. Moreover, this component is especially useful when the variable names for plotting are unseen in the training set, thus NL-code linking provides the only cue to indicate that these variables are relevant.

4.2 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 2, the program sketch does not include the plotted data. After that, the decoder predicts the plotted data, where it employs a copy mechanism (Gu et al., 2016; Vinyals et al., 2015) to select tokens from the code context.

First, we initiate the hidden state of LSTM$_p$ with H_c, the final hidden state of LSTM$_c$, and the start token is `[GO]` for both sketch and full program decoding. At each step t, let s_{t-1} and o_{t-1} be the sketch token and output program token generated at the previous step. Note that s_{t-1} and o_{t-1} are different only when $s_{t-1} \in \{\texttt{VAR}, \texttt{DF}, \texttt{STR}\}$, where o_{t-1} is the actual data name with the corresponding type. Let es_{t-1} and eo_{t-1} be the embedding vectors of s_{t-1} and o_{t-1} respectively, which are computed using the same embedding matrix for the code context encoder. The input of LSTM$_p$ is the concatenation of the two embedding vectors, i.e., $[es_{t-1}; eo_{t-1}]$.

Attention. To compute attention vectors over the natural language description and the code context, we employ the two-step attention in (Iyer et al., 2018). Specifically, we first use hp_t to compute the attention vector over the natural language input using the standard attention mechanism (Bahdanau et al., 2015), and we denote the attention vector as attn$_t$. Then, we use attn$_t$ to compute the attention vector over the code context, denoted as attp$_t$.

Sketch decoding. For sketch decoding, the model computes the probability distribution among all sketch tokens in the code token vocabulary V_c:

$$Pr(s_t) = \text{Softmax}(W_s(hp_t + \text{attn}_t + \text{attp}_t))$$

Here W_s is a linear layer. For hierarchical decoding, we do not allow the model to directly decode the names of the plotted data during sketch decoding, so s_t is selected only from the valid sketch tokens, such as library names, plotting function names, and special tokens for plotted data representation in templates discussed in Section 3.2.

Data selection. For $s_t \in \{\texttt{VAR}, \texttt{DF}, \texttt{STR}\}$, we use the copy mechanism to select the plotted data from the code context. Specifically, our decoder includes 3 pointer networks (Vinyals et al., 2015) for selecting data with the type VAR, DF, and STR respectively, and they employ similar architectures but different model parameters.

We take variable name selection as an instance to illustrate our data selection approach using the copy

Split	Train	Dev (gold)	Test (gold)	Dev (hard)	Test (hard)
All	38971	57	48	827	894
Scatter	11895	19	17	254	276
Hist	8856	14	11	182	175
Pie	574	1	1	14	13
Scatter+Plot	1533	3	1	34	57
KDE	2609	3	5	51	64
Others	13504	17	13	292	309

Table 1: Dataset statistics. The description of the different plot categories is in Section 3.3.

mechanism. We first compute $v_t = W_v(\text{attn}_t)$, where W_v is a linear layer. For the i-th token c_i in the code context, let hc_i be its embedding vector, we compute its prediction probability as:

$$Pr(c_i) = \frac{\exp v_t^T hc_i}{\sum_j \exp v_t^T hc_j}$$

After that, the model selects the token with the highest prediction probability as the next program token o_t, and uses the corresponding embedding vectors for s_t and o_t as the input for the next decoding step of LSTM_p.

The decoding process terminates when the model generates the `[EOF]` token.

5 Experiments

In this section, we first describe our dataset for visualization code synthesis, then introduce our experimental setup and discuss the results.

5.1 Dataset Construction

We build our benchmark upon the JuiCe dataset, and select those that call plotting APIs, including those from `matplotlib.pyplot` (`plt`), `pandas.DataFrame.plot`, `seaborn` (`sns`), `ggplot`, `bokeh`, `plotly`, `geoplotlib`, `pygal`. Over 99% of the samples use `plt`, `pandas.DataFrame.plot`, or `sns`. We first extract plot samples from the original dev and test splits of JuiCe to construct *Dev (gold)* and *Test (gold)*. However, the gold splits are too small to obtain quantitative results. Therefore, we extract around 1,700 Jupyter notebooks of homeworks and exams from JuiCe's training set, and split them roughly evenly into *Dev (hard)* and *Test (hard)*. All remaining plot samples from the JuiCe training split are included in our training set. The length of the visualization programs to be generated varies between 6 and 80 tokens, but the code context is typically much longer. We summarize the dataset statistics in Table 1.

5.2 Evaluation Setup

Implementation details. Unless otherwise specified, for the input specification we include $K = 3$ previous code cells as the local context, which usually provides the best accuracy. We set 512 as the length limit for both the natural language and the code context. For all model architectures, we train them for 50 epochs, and select the best checkpoint based on the program accuracy on the *Dev (hard)* split. More details are deferred to Appendix A.

Baselines. We compare the full PLOTCODER against the following baselines: (1) - *Hierarchy*: the encoder is the same as in the full PLOTCODER, but the decoder directly generates the full program without predicting the sketch. (2) - *Link*: the encoder does not use NL-code linking, and the decoder is not hierarchical. (3) *LSTM*: the model does not use NL-code linking, copy mechanism, and hierarchical decoding. The encoder still uses two separate LSTMs to embed the natural language and code context, which performs better than the LSTM baseline in prior work (Agashe et al., 2019). (4) + *BERT*: we use the same hierarchical decoder as the full model, but replace the encoder with a Transformer architecture (Vaswani et al., 2017) initialized from a pre-trained model, and we fine-tune the encoder with other part of the model. We evaluated two pre-trained models. One is RoBERTa-base (Liu et al., 2019), an improved version of BERT-Base (Devlin et al., 2018) pre-trained on a large text corpus. Another is codeBERT (Feng et al., 2020), which has the same architecture as RoBERTa-base, but is pre-trained on GitHub code in several programming languages including Python, and has demonstrated good performance on code retrieval tasks. To demonstrate the effectiveness of target code canonicalization discussed in Section 3.2, we also compare with models that are directly trained on the raw ground truth code from the same set of Jupyter notebooks.

5.3 Results

We present the program prediction accuracies in Table 2. First, training on the canonicalized code significantly boosts the performance for all models, suggesting that canonicalization improves data quality and hence prediction accuracies. When trained with target code canonicalization, the full PLOTCODER significantly outperforms other model variants on different data splits. 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 when trained on canonicalized target code. 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, especially on plotted data accuracies, as shown in Table 3.

Interestingly, while our hierarchical decoding, NL-code linking, and copy mechanism are mainly designed to improve the prediction accuracy of the plotted data, as shown in Table 4, we observe that the plot type accuracies of our full model are also mostly better, especially on the hard splits. To better understand this, we break down the results by plot type, and observe that the most significant improvement comes from the predictions of scatter plots ("S") and plots in "Others" category. We posit that these two categories constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples. In addition, we observe that the full model does not always perform better than other baselines on data splits of small sizes, and the difference mainly comes from the ambiguity in the natural language description. We defer more discussion to Section 5.4.

Also, using BERT-like encoders does not improve the results. This might be due to the difference in data distribution for pre-training and vocabularies. Specifically, RoBERTa is pre-trained on English passages, which does not include many visualization-related descriptions and code comments. Therefore, the subword vocabulary utilized by RoBERTa breaks down important keywords for visualization, e.g., "scatterplots" and "histograms" into multiple words, which limits model performance, especially for plot type prediction. Using codeBERT improves the performance of RoBERTa, but it still does not improve over the LSTM-based models, which may again due to vocabulary mismatch. As a result, in Table 4, the plot type accuracies of both models using BERT-like encoders are considerably lower than the LSTM-based models.

To better understand the plotted data prediction performance, in addition to the default plotted data accuracy that requires the data order to be the same as the ground truth, we also evaluate a relaxed

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
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%
Without code canonicalization				
Full Model	20.58%	22.73%	22.92%	28.07%
− Hierarchy	20.25%	22.85%	18.75%	26.32%
− Link	20.02%	21.77%	20.83%	24.56%
LSTM	16.22%	16.93%	16.67%	24.56%
+ CodeBERT	20.92%	22.61%	22.92%	24.56%
+ RoBERTa	20.47%	22.37%	20.83%	24.56%

Table 2: Evaluation on program accuracy.

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
Full Model	**40.16%**	**38.69%**	**60.42%**	**49.12%**
− Hierarchy	35.91%	37.00%	47.92%	47.37%
− Link	35.46%	35.67%	47.92%	47.37%
LSTM	29.87%	28.05%	43.75%	40.35%
+ codeBERT	38.14%	38.33%	58.33%	40.35%
+ RoBERTa	37.47%	38.33%	58.33%	29.82%
Without code canonicalization				
Full Model	24.94%	27.69%	29.17%	33.33%
− Hierarchy	26.73%	27.93%	31.25%	31.58%
− Link	25.39%	27.21%	25.00%	28.07%
LSTM	18.90%	21.04%	18.75%	26.32%
+ CodeBERT	26.85%	27.21%	29.17%	31.58%
+ RoBERTa	25.28%	27.81%	27.08%	28.07%

Table 3: Evaluation on plotted data accuracy.

version without ordering constraints. Note that the ordering includes two factors: (1) the ordering of the plotted data for the different axes; and (2) the ordering of plots when multiple plots are included. We observe that the ordering issue happens for around 1.5% of samples, and is more problematic for scatter plots ("S") and "Others." Figure 3 shows sample predictions where the model selects the correct set of data to plot, but the ordering is wrong. Although sometimes the natural language explicitly specifies which axes to plot (e.g., Figure 3 (a)), such descriptions are mostly implicit (e.g., Figure 3 (b)), making it hard for the model to learn. Full results on different plot types are in Section 5.4.

5.3.1 The Effect of Different Model Inputs

To evaluate the effect of including different input specifications, we present the results in Table 5. Specifically, - *NL* means the model input does not include the natural language, and - *Distant DFs* means the code context only includes the local code cells. Interestingly, even without the natural language description, PLOTCODER correctly predicts a considerable number of samples. Figure 4 shows sample correct predictions without relying on the natural language description. To predict the plotted

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
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%
Without code canonicalization				
Full Model	63.53%	65.66%	72.92%	80.70%
− Hierarchy	61.41%	67.47%	66.67%	73.68%
− Link	61.30%	63.72%	64.58%	77.19%
LSTM	64.65%	65.78%	81.25%	70.18%
+ CodeBERT	56.04%	57.07%	60.42%	56.14%
+ RoBERTa	61.30%	61.91%	68.75%	49.12%

Table 4: Evaluation on plot type accuracy.

(a) Natural Language

Create a scatter plot of the observations in the 'credit' dataset for the attributes 'Duration' and 'Age' (age should be shown on the xaxis).

Local Code Context

```
duration = credit['Duration'].values
age = credit['Age'].values
```

Ground Truth

```
plt.scatter(age, duration)
```

Prediction

```
plt.scatter(duration, age)
```

(b) Natural Language

This graph provides more evidence that the higher a state's participation rates, the lower that state's averages scores are likely to be. The higher the participation rate, the lower the expected average verbal scores.

Local Code Context

```
plt.plot(sat_data['Math'], sat_data['Verbal'])
```

Dataframe Schema

```
sat: ['Rate', 'Math', 'Verbal']
```

Ground Truth

```
plt.plot(sat_data['Rate'], sat_data['Math'])
plt.plot(sat_data['Rate'], sat_data['Verbal'])
```

Prediction

```
plt.plot(sat_data['Math'], sat_data['Verbal'])
plt.plot(sat_data['Rate'], sat_data['Verbal'])
```

Figure 3: Examples of predictions where the model selects the correct set of data to plot, but the order is wrong.

data, a simple yet effective heuristic is to select variable names appearing in the most recent code context. This is also one possible reason that causes the wrong data ordering prediction in Figure 3(a); in fact, the prediction is correct if we change the order of assignment statements for variables `age` and `duration` in the code context.

Input	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
Full input	**34.79%**	**34.70%**	**56.25%**	**47.37%**
− Distant DFs	34.34%	34.10%	52.08%	45.61%
− NL	27.52%	28.42%	43.75%	21.05%

Table 5: Evaluation on the full hierarchical model with different inputs.

Meanwhile, we evaluated PLOTCODER by varying the number of local code cells K. The results show that the program accuracies converge or start

(a) Natural Language

Plot a Gaussian by looping through a range of x values and creating a resulting list of Gaussian values, g

Local Code Context

```
x_axis = np.arange(-20, 20, 0.1)
g = []
for x in x_axis:
    g.append(f(mu, sigma2, x))
```

Ground Truth & Prediction

```
plt.plot(x_axis, g)
```

(b) Natural Language

Like in Q9, let's start by thinking about two dice

Local Code Context

```
results = []
for i in range(1,7):
    for j in range(1,7):
        print((i,j),max(i,j))
        results.append(max(i,j))
```

Ground Truth & Prediction

```
plt.hist(results)
```

Figure 4: Examples of model predictions even without the natural language input.

Natural Language

Problem 5. Age groups (1 point) Create a histogram of all people's ages. Use the default settings. Add the label "Age" on the x-axis and "Count" on the y-axis.

Local Code Context

```
income_data.columns = ["age","workclass","fnlwgt",
"education","education_num", "marital_status",
"occupation","relationship","race","sex",
"capital_gain","capital_loss","hours_per_week",
"native_country","income_class"]
...
married_af_peoples = \\
income_data[income_data["marital_status"].str.contains(
"Married-AF-spouse")].shape[0]
...
```

Dataframe Schema

```
income_data: ['age', 'workclass', ..., 'income_class']
married_af_peoples: ['age', 'workclass', ..., 'income_class']
```

Ground Truth

```
plt.hist(income_data.age)
```

Prediction

```
plt.hist(married_af_peoples.age)
```

Figure 5: A sample prediction that requires a good understanding of the code context.

to decrease when $K > 3$ for different models, as observed in (Agashe et al., 2019). However, the accuracy drop of our hierarchical model is much less noticeable than the baselines, suggesting that our model is more resilient to the addition of irrelevant code context. See Appendix B for more discussion.

5.4 Prediction Results Per Plot Type

We present the breakdown results per plot type in Tables 6 and 7. To better understand the plotted data prediction performance, in addition to the default plotted data accuracy that requires the data order to be the same as the ground truth, we also evaluate a relaxed version without ordering constraints, described as *permutation invariant* in Table 7. We compute the results on Test (hard), which

has more samples per plot type than the gold splits. Compared to the non-hierarchical models, the most significant improvement comes from the predictions of scatter plots ("S") and plots in "Others" category. We posit that these two categories constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples. The accuracy of the hierarchical model on some categories is lower than the baseline's, but the difference is not statistically significant since those categories only contain a few examples. A more detailed discussion is included in Appendix C.

Model	S	H	Pie	S+P	KDE	Others
With code canonicalization						
Full Model	**77.17%**	70.86%	61.54%	12.28%	29.69%	**84.14%**
− Hierarchy	70.65%	68.00%	**76.92%**	15.79%	**39.06%**	71.20%
− Link	73.55%	68.00%	69.23%	**21.05%**	35.94%	70.55%
LSTM	73.91%	**71.43%**	69.23%	**21.05%**	28.13%	73.79%
+ codeBERT	67.39%	66.29%	**76.92%**	**21.05%**	35.94%	77.02%
+ RoBERTa	61.59%	62.29%	61.54%	10.53%	34.38%	80.58%
Without code canonicalization						
Full Model	71.01%	74.29%	76.92%	12.28%	37.50%	65.05%
− Hierarchy	75.00%	72.00%	61.54%	14.04%	31.25%	58.25%
− Link	72.10%	60.57%	69.23%	22.81%	37.50%	63.75%
LSTM	74.64%	74.29%	69.23%	19.30%	29.69%	65.70%
+ codeBERT	71.01%	56.00%	46.15%	14.04%	35.94%	55.02%
+ RoBERTa	73.91%	47.13%	46.15%	10.53%	29.69%	74.43%

Table 6: Plot type accuracy on Test (hard) per type.

Model	All	S	H	Pie	S+P	KDE	Others
Plotted data accuracy							
Full Model	**40.16%**	**42.39%**	**41.14%**	61.54%	**10.53%**	**21.88%**	**45.95%**
− Hierarchy	35.91%	35.87%	40.00%	**69.23%**	8.77%	**21.88%**	40.13%
− Link	35.46%	36.96%	39.43%	53.85%	8.77%	14.06%	40.45%
LSTM	29.87%	30.43%	33.14%	61.54%	8.77%	12.50%	33.66%
+ codeBERT	38.14%	38.41%	39.43%	61.54%	8.77%	20.31%	44.98%
+ RoBERTa	37.47%	39.13%	36.57%	**69.23%**	3.51%	17.19%	45.63%
Plotted data accuracy (permutation invariant)							
Full Model	**41.50%**	**44.57%**	**41.14%**	61.54%	**12.28%**	**21.88%**	**47.57%**
− Hierarchy	37.47%	38.04%	40.00%	**69.23%**	10.53%	**21.88%**	42.39%
− Link	41.05%	40.58%	39.43%	53.85%	8.77%	15.62%	43.04%
LSTM	30.65%	31.88%	33.14%	61.54%	10.53%	12.50%	34.30%

Table 7: Plotted data accuracy on Test (hard) per type. All models are trained with canonicalized target code.

5.4.1 Error Analysis

To better understand the challenges of our task, we conduct a qualitative error analysis and categorize the main reasons of error predictions. We investigate all error cases on *Test (gold)* split for the full hierarchical model, and present the results in Table 8. We summarize the key observations below, and defer more discussion to Appendix E.

- Around half of error cases are due to the ambiguity of the natural language description. (1-3)
- About 10% samples require longer code context for prediction, because the program selects the plotted data from distant code context that exceeds the input length limit. (4)
- Sometimes the model generates semantically same but syntactically different programs from the ground truth, which can happen when two variables or data frames contain the same data.(5)
- Besides understanding complex natural language description, as shown in Figure 3, another challenge is to understand the code context and reason about the data stored in different variables. For example, in Figure 5, although both dataframes `income_data` and `married_af_peoples` include the `age` column, the model must infer that `married_af_peoples` is a subset of `income_data`, thus it should select `income_data` to plot the statistics of people from all groups. (6-7)

Error Category	%
(1) NL only suggests the plot type	28.57
(2) NL only suggests the plotted data	9.52
(3) NL has no plotting information	9.52
(4) Need more code context	9.52
(5) Semantically correct	14.29
(6) Challenging NL understanding	19.05
(7) Challenging code context understanding	9.52

Table 8: Error analysis on *Test (gold)* with the hierarchical model.

6 Conclusion

In this paper, we conduct the first study of visualization code synthesis from natural language and programmatic context. We describe PLOTCODER, a model architecture that includes an encoder that links the natural language description and code context, and a hierarchical program decoder that synthesizes plotted data from the code context and dataframe items. Results on real-world Jupyter notebooks show that PLOTCODER can synthesize visualization code for different plot types, and outperforms various baseline models.

Acknowledgments

This material is in part based upon work supported by the National Science Foundation under Grant No. TWC-1409915, IIS-1546083, IIS-1955488, IIS-2027575, CCF-1723352, DOE award DE-SC0016260, DARPA D3M under Grant No. FA8750-17-2-0091; Berkeley DeepDrive, the Intel-NSF CAPA center, and gifts from Adobe, Facebook, Google, and VMware. Xinyun Chen is supported by the Facebook Fellowship.

References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5439–5449.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Victor Dibia and Çağatay Demiralp. 2019. Data2vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks. *IEEE computer graphics and applications*, 39(5):33–46.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742.

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, pages 1–12. ACM.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

John D Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90–95.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4159–25.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv:1907.11692 [cs]*. ArXiv: 1907.11692.

Matplotlib. 2020. Matplotlib scatter method documentation. https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.scatter.html.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE.

Pandas. 2020. Pandas dataframe documentation. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428.

Seaborn. 2020. mwaskom/seaborn library documentation. https://doi.org/10.5281/zenodo.592845.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. thesis, UC Berkeley.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in neural information processing systems*, pages 2692–2700.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2019a. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 452–466. ACM.

Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019b. Visualization by example. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28.

Chenglong Wang, Yu Feng, Rastislav Bodík, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-powered visualization authoring. In *CHI '21: CHI Conference on Human Factors in Computing Systems*, pages 106:1–106:15. ACM.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, et al. 2019a. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, et al. 2019b. Sparc: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523.

Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. 2018. Naps: Natural program synthesis dataset. *arXiv preprint arXiv:1807.03168*.

Luke S Zettlemoyer and Michael Collins. 2012. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

A Implementation Details

For the model input, we select the suffix of the code sequence when it exceeds the length limit, and we select the prefix for the natural language. To construct the vocabularies, we include natural language words that occur at least 15 times in the training set, and code tokens that occur at least 1,000 times, so that each vocabulary includes around $10,000$ tokens. We include an `[UNK]` token in both vocabularies, which is used to encode all input tokens not appeared in our vocabularies.

The model parameters are randomly initialized within $[-0.1, 0.1]$. Each LSTM has 2 layers, and a hidden size of 512. The embedding size of all embedding matrices is 512, and the hidden size of the linear layers is 512. For training, the batch size is 32, the initial learning rate is `1e-3`, with a decay rate of 0.9 after every $6,000$ batch updates. The dropout rate is 0.2, and the norm for gradient clipping is 5.0.

For models using the Transformer architecture as the encoder, we use the pre-trained RoBERTa-base and codeBERT from their official repositories.[2] The hyper-parameters are largely the same as the LSTM-based models, except that we added a linear learning rate warmup for the first $6,000$ training steps, which is the common practice for fine-tuning BERT-like models.

B Training with Varying Number of Contextual Code Cells

As discussed in Section 5.3.1, we provide the results of including different number of local code cells as the model input in Figure 6. We also evaluated the upper bounds of program accuracies for different values of K, where we consider an example to be predictable if all plotted data in the target program are covered in the input code context. We observe that including dataframe manipulation code in distant code cells improves the coverage, especially when K is small.

C Detailed Analysis on Results Per Plot Type

In Section 5.4, we present the breakdown results per plot type in Tables 6 and 7, where we observed that "Scatter" and "Others" constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples.

Note that for categories that the hierarchical model does not perform better than baselines, even if the accuracy differences are noticeable, the numbers of correct predictions do not differ much. For example, among the 13 samples in the "Pie" category, the hierarchical model correctly classifies 8 samples, while the non-hierarchical version makes 10 correct predictions. When looking at the predictions, we observe that the 2 different predictions are mainly due to the ambiguity of the natural language descriptions. Specifically, the text descriptions are "The average score of group A is better than average score of group B in 51% of the state" and "I am analyzing the data of all male passengers". In fact, for both examples, the hierarchical model still generates a program including the plotted data in the ground truth. However, the hierarchical model wrongly selects `plt.bar` as the plotting API for the former sample, and selects `plt.scatter` for the latter sample, where it additionally selects another variable for the x-axis. For these 2 samples, we observe that the code context includes plotting programs that use other data to generate pie charts,

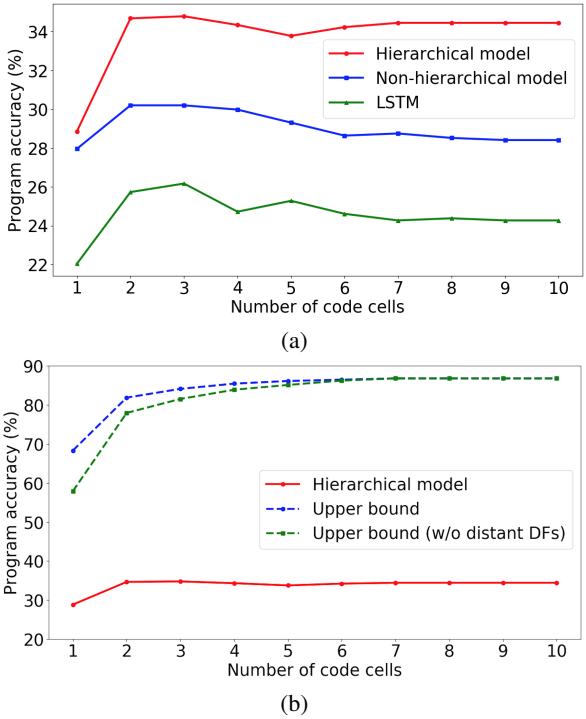

Figure 6: Program accuracy with different number of input code cells. (a) Results of different model architectures. (b) The comparison between the accuracy of the hierarchical model and the upper bounds.

[2] RoBERTa: https://github.com/pytorch/fairseq/tree/master/examples/roberta
codeBERT: https://github.com/microsoft/CodeBERT

and the non-hierarchical model picks a heuristic to select the same plot type in the code context when there is no cue provided in the natural language description, while the hierarchical model selects plot types that happen more frequently in the training distribution. A similar phenomenon holds for other categories or data splits with a small number of examples.

D Other Plot Types

In the "Others" category discussed in Section 3.3, besides the plots generated by `plt.plot`, there are also other plot types, with much smaller data sizes than `plt.plot`. In Table 9, we present the breakdown accuracies of some plot types, which constitute the largest percentages in the "Others" category excluding `plt.plot` samples. Specifically, around 4% samples use `boxplot`, and each of the other 3 plot types include around 1% samples. Due to the lack of data for such plot types, the results are much lower than the overall accuracies of all plot categories, but still non-trivial.

Plot Type	Plot Type Acc	Plotted Data Acc	Program Acc
boxplot	51.04%	10.42%	7.29%
pairplot	42.31%	34.62%	23.00%
jointplot	36.36%	9.09%	4.55%
violinplot	47.06%	5.88%	5.88%

Table 9: Breakdown accuracies of plots in "Others" category on Test (hard), using the full hierarchical model.

E More Discussion of Error Analysis

As discussed in Section 5.4.1, the lack of information in natural language descriptions is the main reason for a large proportion of wrong predictions (categories 1-3 in Table 8).

- Many natural language descriptions only mention the plot type, e.g., "Make a scatter plot," which is one reason that the plot type accuracy is generally much higher than the plotted data accuracy. (1)
- Sometimes the text only mentions the plotted data without specifying the plot type, e.g., "Plot the data x_1 and x_2," where both `plt.plot(`x_1`,`x_2`)` and `plt.scatter(`x_1`,`x_2`)` are possible predictions, and the model needs to infer the plot type from the code context. (2)
- The text description includes no plotting information at all, e.g., "Localize your search around the value you found above," where the model needs to infer which variables are search results and could be plotted. (3)

We consider several directions to address different error categories as future work. To mitigate the ambiguity of natural language descriptions, we could incorporate additional program specifications such as input-output examples. Input-output examples are also helpful for evaluating the execution accuracy, which considers all semantically correct programs as correct predictions even if they differ from the ground truth. Most Jupyter notebooks from GitHub do not contain sufficient execution information, e.g., many of them load external data for plotting, and the data sources are not public. Therefore, developing techniques to automatically synthesize input-output examples is a promising future direction. Designing new models for code representation learning is another future direction, which could help address the challenge of embedding long code context.