HAConvGNN: Hierarchical Attention Based Convolutional Graph Neural Network for Code Documentation Generation in Jupyter Notebooks

Xuye Liu ‡
University of Waterloo

Dakuo Wang ‡
IBM Research

April Yi Wang
University of Michigan

Yufang Hou
IBM Research Europe

Lingfei Wu *
JD.COM Silicon Valley Research Center

Abstract

Jupyter notebook allows data scientists to write machine learning code together with its documentation in cells. In this paper, we propose a new task of code documentation generation (CDG) for computational notebooks. In contrast to the previous CDG tasks which focus on generating documentation for single code snippets, in a computational notebook, one documentation in a markdown cell often corresponds to multiple code cells, and these code cells have an inherent structure. We proposed a new model (HAConvGNN) that uses a hierarchical attention mechanism to consider the relevant code cells and the relevant code tokens information when generating the documentation. Tested on a new corpus constructed from well-documented Kaggle notebooks, we show that our model outperforms other baseline models.

1 Introduction

In recent years, computational notebooks such as Jupyter have become popular programming platforms for data scientists and machine learning researchers to document ideas, write code, and visualize results, all in a single document (Wang et al., 2021a). Documentation in a notebook provides a rich medium for users to record not only what the code does, but also why they code it. This richness of content is one distinctive nature of code documentation in a notebook versus in traditional software source code.

Code documentation is found critical for data scientists to share or reuse code (Zhang et al., 2020; Chattopadhyay et al., 2020). However, research has shown that many data scientists still neglect to write appropriate documentation for their code in notebooks, as they feel writing documentation will slow down their coding process. Rule et al. (2018) report that among one million computational notebooks on Github, 25% of them have no comment.

Table 1: An example of multiple code cells after one documentation block

<table>
<thead>
<tr>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ground truth</td>
</tr>
<tr>
<td>our Model</td>
</tr>
<tr>
<td>code2seq</td>
</tr>
<tr>
<td>graph2seq</td>
</tr>
<tr>
<td>TS-small</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>import keras</td>
</tr>
<tr>
<td>from keras.utils import plot_model</td>
</tr>
<tr>
<td>from keras.models import</td>
</tr>
<tr>
<td>↪ Model, Sequential, load_model</td>
</tr>
</tbody>
</table>

...  

```python
def nn_model(X,y,optimizer,kernels):
    input_shape = X.shape[1]
    if(len(np.unique(y)) == 2):
        op_neurons = 1
        op_activation = 'sigmoid'
        loss = 'binary_crossentropy'
    else:
        op_neurons = len(np.unique(y))
        op_activation = 'softmax'
        loss = 'categorical_crossentropy'
    classifier = Sequential()
    ...
    classifier.summary()
    return classifier
```

model =  
```python
↪ nn_model(X_train,y_train,'adam','he_uniform')
history = model.fit(X_train, y_train,
↪ batch_size = 64, epochs = 1000,
↪ validation_data=(X_test, y_test))
```

```python
pd.DataFrame(abs(train.corr()['Survived']).sort_values(ascending = False))
```

As a first step towards building an automated documentation generation system for notebooks, in this paper we focus on the code documentation generation (CDG) task for Jupyter notebooks. Since there is no publicly available CDG dataset for notebooks, we construct a new dataset (notebookCDG) which contains around 28k processed code-documentation pairs extracted from 2,476 highly-ranked notebooks from Kaggle competitions (details in Section 3)

Part of work was done when Xuye, April, and Lingfei were at IBM.
A few previous literature have explored techniques to generate documentation for software code snippet one at a time (LeClair et al., 2020; Haque et al., 2020, 2021; Xu et al., 2018). However, in computational notebooks, one documentation (in a markdown cell) can cover more-than-one code cells after it. For instance, the ground truth text in Table 1 is a single documentation covering four code cells. Existing work on CDG (Kery and Myers, 2017; Iyer et al., 2016; Hu et al., 2018; Alon et al., 2019; LeClair et al., 2020) does not consider such structure information since they only focus on documentation generation for single code snippet (i.e., one function, or one expression).

To account for the above mentioned properties of documentation in computational notebooks, in this paper, we propose a graph-augmented encoder-decoder model to generate documentation for notebooks (Section 4). In particular, our model consists of three parts: a code sequence encoder, an auxiliary documentation text encoder based on the already predicted documentation tokens, and a Hierarchical Attention-based Convolutional Graph Neural Network (HAConvGNN) component.

The first two sequence encoders encode the semantic information in code and documentation text, respectively. The graph encoder encodes the contextual abstract syntactic trees (i.e., AST extracted from the code sequence). In order to capture the relations between code sequences and the corresponding text documentations, we further employ a hierarchical attention mechanism consisting of a low-level attention module and a high-level attention module. The former attends to the token in a code sequence and the latter attends to the corresponding code cells in the AST tree.

Experiments show that our model achieves better performance on the notebookCDG dataset compared to baseline models on ROUGE scores, and in a multi-dimensional human evaluation study.

Base on this result, we integrated our approach into a user-facing downstream application (Wang et al., 2021c) to further explore the Human-AI collaboration opportunity in the code documentation scenario. In the follow-up user study (reported separately (Wang et al., 2021b)), users found that the automatically generated documentation reminded them to document code they would have ignored, and improved their satisfaction with their computational notebooks.

In summary, the main contributions of our work are: (1) a large-scale high quality dataset for the CDG task in the computational notebook context; (2) a graph-based neural network architecture with hierarchical attention for the notebook CDG task which considers the structure information between multiple code cells and the relations between code tokens and text tokens; and (3) human evaluations to validate our model for real world application. The experiment code and data are shared\textsuperscript{1}.

## 2 Related Work

In order to automate the machine learning and AI workflow, researchers have applied automation techniques on various code-related tasks (Wang et al., 2020), including code summarization (Iyer et al., 2016; LeClair et al., 2020; Haque et al., 2020, 2021), source code generation from natural language (Agashe et al., 2019), and source code transformation (Roziere et al., 2020).

In this work, we focus on the code documentation generation (CDG) task. Our work is closely related to code summarization. Most existing datasets for code summarization contain one summary per one code snippet. For instance, CodeSearchNet (Husain et al., 2019) contains two million function-documentation pairs across six programming languages (e.g., java, php, python). In contrast, our new dataset (notebookCDG) is designed for computational notebooks. The difference from previous CDG datasets is that in our dataset, a documentation text can correspond to several code snippets.

Previous work on code summarization focuses on summary generation for a single standalone code snippet. Iyer et al. (2016) collected Stack Overflow question titles as code summaries and paired them with top-rated code snippets. They then used an attention seq2seq model to generate a summary for each code snippet. Several studies explored the abstract syntactic tree (AST) information of source code to better capture the relation between different elements (Hu et al., 2018; Alon et al., 2019). Recently, Xu et al. (2018) and Chen et al. (2020) have proposed a general graph to sequence model to learn node embeddings and then reassemble them into the graph embeddings.

Unlike the aforementioned works that only focus on summary generation for a single standalone code snippet, in our new CDG task for computational notebooks, multiple adjacent code cells can

\textsuperscript{1}https://github.com/dakuo/HAConvGNN
Table 2: notebookCDG dataset statistics. The overall code-to-markdown ratio is 2.2195, which suggests one markdown corresponds to more than one code cells.

correspond to one documentation and these code cells may have a hierarchical structure, and use a graph to represent it (Kipf and Welling, 2016). We thus propose Hierarchical Attention-based Convolutional Graph Neural Network (HAConvGNN) to handle the hierarchical AST graph structure of multiple code cells.

3 notebookCDG Dataset

CDG for notebooks is a relatively new task. To our best knowledge, we could not find an appropriate dataset for this task. Thus, we decided to construct a new dataset and share it with the community. Publicly shared notebooks on Github are often ill-documented (Rule et al., 2018), thus are not suitable for constructing the training dataset for CDG task. A recent work (Wang et al., 2021a) manually analyzed 80 publicly available notebooks on two Kaggle challenges (i.e. out of 12,000 notebooks submitted to Titanic and HousePrice). Kaggle allows community members to vote up and down on those notebooks, and Wang et al. (2021a)’s findings show that the highly-voted notebooks are of good quality and quantity in code documentation. Inspired by their work, we decided to utilize the top-voted and well-documented Kaggle notebooks to construct the notebookCDG dataset.

We collected the top 10% highly-voted notebooks from the top 20 popular competitions on Kaggle (e.g. Titanic). We checked the data policy of each of the 20 competitions, none of them has copyright issues. We also contacted the Kaggle administrators to make sure our data collection complies with the platform’s policy. In total, we collected 3,944 notebooks as raw data.

3.1 Data Preprocessing

We performed various preprocessing steps to prepare the dataset, following LeClair and McMillan (2019). For example, we removed notebooks in non-English language. One major difference between our dataset and previous datasets is that in previous datasets, each documentation unit is corresponding to one code snippet, whereas in our dataset, one documentation unit may correspond to up to four code snippets (code cells). We first located the markdown cells that have code cells beneath them. According to Wang et al. (2021a), there are nine categories of documentations in a notebook, some are related to code, some are not related to code. For those types closely related to code (Process and Headline), which take up 80% of the cases, we can directly use the markdown cell as documentation. For some other types, such as the Result type, which interprets the rendered result table or plot thus are often long and irrelevant to the code, we used a list of keywords (e.g., shows) to filter out the key sentences from the markdown cell as the documentation. Another special types of documentation are Reason and Education, which also uses long word sequence...
to explain why the author did something. In these cases, based on our observation, we used the first sentence as the documentation, as the first sentence is often related to the code cells.

Our analysis shows that for one markdown cell, there could have maximum four code cells following it. We construct our dataset to have a structure with one documentation unit and four code sequence units, and fill with empty sequence if there is less than four code sequences. As part of the data preparation, we also parse each of code sequence to an AST graph structure through a Python AST library. While doing so, we removed all the non-Python notebook magic (e.g. %matplotlib).

3.2 Dataset Core Statistics
After data preprocessing, the final dataset contains 2,476 notebooks out of the 3,944 notebooks from the raw data. It has 28,625 code–documentation pairs. The overall code-to-markdown ratio is 2.2195, which suggests one markdown corresponds to more than one code cells. Then, the code-documentation pairs are randomly split into train, dev, and test subsets, following a 8:1:1 ratio (Table 2).

Our notebookCDG dataset has a vocabulary size of 13,053 for the documentation sequence, a vocabulary size of 20,522 for the code sequence, and 67,211 for the parsed code AST node. On average, each pair of code-documentation has 65.38 code tokens, and 9.15 documentation tokens. When code is translated to AST structure, on average it has 181.08 tokens.

4 Approach
Our model is built upon the standard encoder-decoder structure. To handle multiple code cells in computational notebooks, we propose a hierarchical attention mechanism based on convolutional graph neural network (HAConvGNN) for capturing the relevant code cells during the decoding stage.

The system architecture is illustrated in Figure 1. Below, we describe each module in detail.

4.1 Model Input
As mentioned in Section 3, we found that there are up to four adjacent code cells under a markdown cell, thus we constructed the notebookCDG dataset to have one documentation mapping to four code cells, and used empty code cell as padding. Therefore, when generating the abstract syntactic tree (AST) for a code cell, we can assemble up to four AST trees into a higher level graph structure.

In summary, each training data point has four parts: the tokenized code sequence, the tokenized documentation sequence, the nodes of the AST graph generated from the code sequence, and the edges (topology) of the AST graph generated from the code sequence. We denoted code sequence input as \( S = \{ s_1, s_2, ..., s_n \} \in S \) where \( s_i \) is sequence consisting of a sequence of code token embeddings \( s_i = \{ w_1, w_2, ..., w_k \} \in W \) in which \( W \) is the token embedding space and \( k \) is the length of \( s_i \). Next we construct the AST graph input \( A = (V, E) \) where \( V \) are the nodes containing the original code, \( E \) are the edges which denote whether two nodes are connected or not in the AST graph.

4.2 Embeddings
We use three embedding layers to generate embeddings for the tokenized code sequence, the nodes in an AST graph, and the documentation decoder, respectively.

4.3 Encoder
We use one encoder to encode the source code sequence, and additional four encoders to encode up to four code cells’ AST graphs. In addition, we have a high-level GRU encoder layer for all the four AST graphs to generate one high-level output. More specifically, the encoder for the tokenized code sequence is a GRU with an output length of 256. An AST graph encoder is a collection of Convolutional Graph Neural Networks layers followed by a GRU layer of output length 256. We use four AST graph encoders for up to four code cells. Following LeClair et al. (2020), the number of hops in our GNN layers is set to 2.

4.4 HAConvGNN
The key design of our HAConvGNN model is the hierarchical attention. When handling AST graphs input, instead of blending these 4 code cells as a whole sequence, we propose to use a hierarchical attention mechanism (low-level attention and high-level attention in HAConvGNN in Figure 1) on these AST graphs to better preserve the graph structure.

Firstly, the four code cells’ AST graph can be represented as \( G = \{ G_1, G_2, G_3, G_4 \} \). We denote
the decoder output (i.e., the predicted documenta-
tion tokens up till $t - 1$) as $D \in \mathbb{R}^{n \times d}$ where $d$ is
the dimension. We further denote each code cell’s
AST graph as $G_i \in \mathbb{R}^{m \times d}$ where $m$ is the number
of nodes. After using a high-level encoder to en-
code the AST graph input, we execute a graph-level
attention to get high-level attention score:
\[
\alpha(G_i, D) = DG_i^\top / \sqrt{d} \tag{1}
\]
Then we apply softmax on $\alpha$, given by:
\[
b_i = \frac{\exp(\alpha(G_i, D))}{\sum_j \exp(\alpha(G_j, D))} \tag{2}
\]
In this way, we get the results denoted as $\alpha = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. This is our high-level attention
weights indicating the relations between each code
cell and the already predicted documentation se-
quence $D$.

Secondly, we apply an attention mechanism on
each code cell to find the relations between nodes in
a code cell’s AST and the predicted documentation
sequence $D$. For each code cell’s AST tree $G = \{G_1, G_2, G_3, G_4\}$, we apply the same operation as
in EQ.1 and EQ.2. As a result, for each code cell
$G_i$, we are able to get a new low-level attention
weight $\beta_i$. For all code cells, we can denote these
attention scores as $\beta = \{\beta_1, \beta_2, ..., \beta_m\}$.

Eventually, we fuse these attention weights ($\alpha$
and $\beta$) with code cells:
\[
O = \sum_{i=1}^{4} \alpha_i \sum_{j=1}^{m} \beta_{i,j} G_{i,j} \tag{3}
\]
Now we get the AST matrices from HACov-
vGNN. It is then concatenated with code matrices
into a single context matrix. Note that code ma-
trices are based on the code sequence input with a
separate uniform attention (see the left “Code
Sequence” in Figure 1). Next, we apply a linear
projection to project the merged context matrix into
a 256 dimension space. This is an effective way
to avoid overfitting during the training process.
Finally, we flatten the new context matrix and apply
another linear layer to project it into an output. The
output layer size is the vocabulary size. By applying
the Argmax function to the output layer, we can
obtain the predicted next token (i.e., documentation
token at time step $t$) in the output sequence.

5 Experimental Setup
5.1 Implementation Details
We split our dataset into training, development, and
test datasets at a 8:1:1 ratio. We use the Adam
optimizer (Kingma and Ba, 2014) with a batch size
of 20. The learning rate is 0.001 and the code
sequence embedding size is 100. In the encoder,
we use GRU (Cho et al., 2014) with the hidden size
of 256. The hop size of our GNN is 2. The dropout
rate of our attention layer is 0.5.

5.2 Baselines
We compare our model against two baseline models
for single code snippets.
code2seq. Alon et al. (2019) proposed a
code2seq model to generate a summary for a C#
function. The model creates a vector representation
for each AST path separately through an encoder.
During decoding, the model uses attention to select
the relevant paths. We re-implement this model and apply it on our dataset.

**graph2seq.** Xu et al. (2018) proposed a graph-to-sequence learning framework that maps an input graph to a sequence of vectors and uses an attention-based LSTM method to decode the target sequence from these vectors. The authors tested the model on natural language question generation from the SQL query task. We re-implement this model using all recommended parameters from the original paper.

### 5.3 Experimental Details

The training time of code2seq model is around 2.5 hours per epoch; the training time of graph2seq is around 2.75 hours per epoch; the training time of T5-small is around 3.25 hours per epoch; the training time of our HAConvGNN model is around 2.65 hours per epoch.

The training environment of code2seq, graph2seq, and HAConvGNN is three GPUs using Parallelism. The training environment of T5-small is two GPUs.

code2seq and graph2seq are implemented in Keras framework\(^4\), T5-small model is implemented based on Huggingface repo \(^5\).

### 6 Automated Evaluation

We use ROUGE scores (Lin, 2004) to evaluate our model’s performance with regard to the ground-truth documentation content. We report ROUGE-1, ROUGE-2, and ROUGE-LCS (longest common sub-sequence). As shown in Table 3, our HAConvGNN model outperforms the other two baselines in all ROUGE metrics.

**Ablation study.** In order to better understand the impact of the attention components in our model, we also perform an ablation study (Table 3). Our ablation study evaluates how low-level attention, high-level attention, and AST uniform attention contribute to the model. More concretely, we generate ablation models as the following:

1. without high-level attention in the hierarchical attention: we remove high level attention component in Figure 1 in our HAConvGNN structure. That means we do not compute attention weights for separated code cells.
2. without AST uniform attention: we do not apply uniform attention mechanism (i.e., the attention component above HAConvGNN in Figure 1 for our HAConvGNN output with the decoder. (3) without low-level or high-level attentions: we

---

**Table 3: ROUGE scores for the baselines, our model, and the ablation models. Results show that our model has higher scores for all three metrics, demonstrating a robust advantage over the code2seq and graph2seq models.**

<table>
<thead>
<tr>
<th>Models</th>
<th>ROUGE-1</th>
<th>ROUGE-2</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F1</td>
</tr>
<tr>
<td>Baselines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>code2seq</td>
<td>11.45</td>
<td>8.46</td>
<td>8.23</td>
</tr>
<tr>
<td>graph2seq</td>
<td>13.21</td>
<td>9.87</td>
<td>9.51</td>
</tr>
<tr>
<td>Our Model &amp; Ablation Study</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAConvGNN (Our Model)</td>
<td>22.87</td>
<td>16.92</td>
<td>16.58</td>
</tr>
<tr>
<td>HAConvGNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with low-level attention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without high-level attention with uniform attention</td>
<td>20.66</td>
<td>15.65</td>
<td>14.91</td>
</tr>
<tr>
<td>HAConvGNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with low-level attention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAConvGNN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without low-level attention</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without high-level attention</td>
<td>11.39</td>
<td>7.73</td>
<td>7.82</td>
</tr>
</tbody>
</table>

---

\(^4\)https://github.com/Attn-to-FC/Attn-to-FC

\(^5\)https://github.com/huggingface/transformers
remove separated low-level attention components in Figure 1) in our HAConvGNN structure. Note that when we remove these separated attentions, we also remove the high-level attention (thus the entire hierarchical attention structure). We treat multiple code cells as a standalone code snippet in this situation and process graph data with the original GNN layer (see the last row in Table 3).

In general, we found that the hierarchical structure in our HAConvGNN is proven to enhance our final performance. It is worth noting that the separated attention mechanism is essential in our model. Remember that we use the attention mechanism for our four code cells separately. Treating them as a single big code snippet leads to a considerable performance drop (see the last row in Table 3). This demonstrates that the hierarchical structure in our model can better handle the code documentation generation task for multiple code cells.

Attention Visualization. Our high-level attention mechanism can indicate the most relevant code cell when generating the documentation for several code cells. Figure 2 illustrates the attention heatmap for the code example in Table 1. Each row represents a code cell, and each column corresponds to a code token. It seems that the modes pays more attention to the second code cell (especially the first few tokens) when generating the documentation “Implementing Neural Network”.

7 Human Evaluation

We also conduct a human evaluation to further evaluate our model against the two baselines and the ground truth.

Participants. Our human evaluation task involves reading code snippets and rating the generated documentation of the codes. We recruited participants with data science and machine learning backgrounds ($N = 15$).

Task. We randomly selected 30 pairs of documentation and code(s) from our dataset. Note that each pair has only one summary, but may have multiple code snippets. Each participant is randomly assigned 10 trials, and the order of these 10 trials is also randomized. Each pair is evaluated by 5 individuals. In each trial, a participant reads 4 candidate documentation for the same code snippet(s): three generated by the three models, and the other one is the groundtruth. Participants do not know which documentation text is from which model. The participant is asked to rate the 4 documentation texts along three dimensions using a five-point Likert-scale from -2 to 2.

- Correctness: The generated documentation matches with the code content.
- Informativeness: The generated documentation covers more information units.
- Readability: The generated documentation is in readable English grammar and words.

Evaluation Results. We conducted pairwise $t$-tests to compare each model’s performance. The result (Table 4) shows that for the Correctness dimension, our model (avg=0.21) is significantly better than the other two baselines (avg=-0.59 for code2seq, avg=-0.30 for graph2seq, both $p<.01$). Our model is also the only model that has a positive rating. For the Informativeness dimension, groundtruth also has the best rating. Our model (avg=0.17) comes in second and outperforms code2seq (avg=-0.72, $p<.01$) and graph2seq (avg=-0.21, $p<.01$).

For the Readability dimension, in which we consider whether generated documentation is a valid English sentence or not, groundtruth outperforms all ML models again, but our model (avg=0.67) also significantly outperforms baseline models code2seq (avg=0.03 $p<.01$) and graph2seq (avg=0.32 $p<.01$). Our model can generate more readable documentation than baselines.
Figure 3: Average rated scores given by human evaluators to each method across three dimensions.

<table>
<thead>
<tr>
<th>Model</th>
<th>Correctness</th>
<th>Informativeness</th>
<th>Readability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groundtruth</td>
<td>$\bar{x} = 1.09, \sigma = 0.95$</td>
<td>$\bar{x} = 0.85, \sigma = 0.97$</td>
<td>$\bar{x} = 1.03, \sigma = 1.01$</td>
</tr>
<tr>
<td>Our model</td>
<td>$\bar{x} = 0.21, \sigma = 1.29$</td>
<td>$\bar{x} = 0.17, \sigma = 1.18$</td>
<td>$\bar{x} = 0.67, \sigma = 1.20$</td>
</tr>
<tr>
<td>Code2seq</td>
<td>$\bar{x} = -0.59, \sigma = 1.40$</td>
<td>$\bar{x} = -0.72, \sigma = 1.17$</td>
<td>$\bar{x} = 0.03, \sigma = 1.35$</td>
</tr>
<tr>
<td>Graph2seq</td>
<td>$\bar{x} = -0.30, \sigma = 1.40$</td>
<td>$\bar{x} = -0.21, \sigma = 1.25$</td>
<td>$\bar{x} = 0.32, \sigma = 1.35$</td>
</tr>
</tbody>
</table>

All the results suggest that our model has above-zero ratings, which suggests it reaches an acceptable user satisfaction along all three dimensions.

8 Comparison With Transformers

We also carried out an additional experiment to compare our model with T5 (Raffel et al., 2020), which is a state-of-the-art transformer encoder-decoder model. In order to fairly compare our model against T5, we do not use any pre-trained embeddings for the T5 model. Also, T5 input has limitation for the input token length thus we did not feed AST hierarchy into it. More specifically, we initialize a T5-small model\(^6\) with random weights and train this model using our training data. Our code adapts the transformer models from HuggingFace (Wolf et al., 2020). We use the dev dataset to choose the hyperparameters and evaluate the trained model on our test dataset. The ROUGE F1 scores for the trained T5-small model are as follows: ROUGE-1 = 17.55, ROUGE-2 = 4.57, ROUGE-L = 19.53.

We found that the trained T5-small model achieves slightly better results than our model in ROUGE-1 and ROUGE-L. In practice, we found that the T5-small model relies on a much more hyperparameters and tends to generate less informative content compared to other models (see the documentation generated from different models in Table 1 for an example).

But in our dataset, as reported in Table 2, the max AST token sequence is 1,732, which is too long as T5 input (512) or BART input (1,024). That is why T5 in Sec 8 can only take the raw code sequence as input, instead of the AST hierarchy. It is known that programming code has a tree-based hierarchy and leveraging such AST hierarchy can enhance the baseline model (e.g., (Alon et al., 2019)). Our contribution is that we provide a hierarchical attention architecture that is well suited for the programming code nature and can generalize to a much longer length of code inputs. Imagine in a scenario where we can feed a whole code repo as training input by treating each code file as a lower layer, and connecting them through function/variable referencing – our architecture can also handle that. In general, we think our model is orthogonal to the standard transformer models. One interesting future work is to integrate our hierarchical attention mechanism into the transformer-based structure instead of a GRU-based structure.

9 Downstream User Application

To demonstrate the application of the HAConvGNN model, we designed a Jupyter Notebook plugin to assist document writing in data science programming (as shown in Figure 4).

The plugin is triggered when detecting users focusing on a code cell (Figure 4.A). The plugin then reads the contents from the focused cell and its adjacent cells, and sends the content to the backend. The backend server first generates a code summarization using the HAConvGNN model (Figure 4.B). In addition, we implemented two other

\(^6\)In a pilot study, training a T5-base model (with random initialization) on our dataset leads to worse results.
approaches to generate documentation that was intended for explaining a design decision or explaining a technical concept for educational purposes. We retrieved the relevant documentation from the API webpage for educational purposes (Figure 4.C) and we used prompts to nudge users to explain an output (Figure 4.D). If the user likes one of these three candidates, they can simply click on one of them, and the selected documentation candidate will be inserted into above the code cell (if it describes what and why for the code), or below it (if it interprets the result of the code).

Our plugin went through several rounds of pilot testing and iterative design. Participants found it reminds them to document code they would have ignored, reduce the time for developing documentation while they were actively exploring the data science task. The implementation details and a formal evaluation of understanding the benefits of the human-AI collaborative effort for automatic documentation generation are reported separately in (Wang et al., 2021b).

10 Conclusion and Future Work

This work targets a new application that aims to automatically generate code documentation (CDG) for a computational notebook. This project is part of our long-term research initiative of designing AI to automate the various tasks in an AI project’s lifecycle (Wang et al., 2021d). The notebookCDG context imposes unique challenges to the current code documentation generation approaches which only consider a single code snippet. We construct a dataset from Kaggle challenge notebooks, and present a novel HAConvGNN model to encode the multiple adjacent code cells as a hierarchical AST graph to enhance a sequence model architecture. Both automated evaluation and human evaluation show that our model outperforms the baseline models. We also incorporate our algorithm into a Jupyter Notebook plugin to assist code documentation creation.

11 Ethical Concern

Our task is an instance of natural language generation task, thus it may have potential risk and ethical issues similar to any other NLG tasks, such as the generated content may have offensive language. However, we believe our task and our approach has minimum risk of such ethical issues, due to two reasons: firstly, the language used in the context of machine learning code documentation is more strict to technical terms, offensive language is less likely to appear in the dictionary thus in our model; secondly, the dataset construction method is to use highly-voted notebooks from a publicly available Kaggle community, there is unlikely to have offensive languages in these highly-voted notebooks.
References


# Appendix: Code snippets-documentation Pair Examples

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Code Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ground Truth</strong></td>
<td>Feature scaling</td>
</tr>
<tr>
<td><strong>Our Model</strong></td>
<td>Feature scaling</td>
</tr>
<tr>
<td><strong>Code2seq</strong></td>
<td>We can have the model</td>
</tr>
<tr>
<td><strong>Graph2seq</strong></td>
<td>The next step is a lot of the training set</td>
</tr>
<tr>
<td><strong>T5-small</strong></td>
<td>Scaling</td>
</tr>
</tbody>
</table>

## Table 5: Example: Feature Scaling
```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(train_df)
train_scale = pd.DataFrame(scaler.transform(train_df))
```

## Table 6: Example: Handle Missing Values
```python
from sklearn.impute import SimpleImputer
my_imputer = SimpleImputer(strategy='most_frequent')
my_imputer.fit(X_train)
imputed_X_test = pd.DataFrame(my_imputer.transform(X_test))
imputed_X_test.columns = X_test.columns
```

## Table 7: Example: Data Augmentation
```python
import warnings
from imgaug import augmenters as iaa
warnings.filterwarnings("ignore")

augmentation = iaa.Sequential([
    iaa.OneOf([# rotate
        iaa.Affine(rotate=0),
        iaa.Affine(rotate=90),
        iaa.Affine(rotate=180),
        iaa.Affine(rotate=270),
    ]),
    iaa.Fliplr(0.5),
    iaa.Flipud(0.2),
    iaa.OneOf([# cutout
        iaa.Cutout(fill_mode="constant", cval=255),
        iaa.CoarseDropout((0.0, 0.05), size_percent=(0.02, 0.25)),
    ]),
    iaa.OneOf([# snowflakes
        iaa.Snowflakes(flake_size=(0.2, 0.4), speed=(0.01, 0.07)),
        iaa.Rain(speed=(0.3, 0.5)),
    ]),
    iaa.OneOf([# contrast
        iaa.Multiply((0.8, 1.0)),
        iaa.contrast.LinearContrast((0.9, 1.1)),
    ]),
    iaa.OneOf([# blur
        iaa.GaussianBlur(sigma=(0.0, 0.1)),
        iaa.Sharpen(alpha=(0.0, 0.1)),
    ]),
], random_order=True)

def get_ax(rows=1, cols=1, size=7):
    _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
    return ax

limit = 4
ax = get_ax(rows=2, cols=limit//2)
for i in range(limit):
    image, image_meta, class_ids,box, mask = modellib.load_image_gt(
        dataset_train, config, image_id,
        use_mini_mask=False,
        augment=False, augmentation=augmentation)
    visualize.display_instances(image, bbox, mask,
        class_ids, dataset_train.class_names,
        ax=ax[1][i], show_bbox=False,
        show_mask=False)
```

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Code Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ground Truth</strong></td>
<td>Data Augmentation</td>
</tr>
<tr>
<td><strong>Our Model</strong></td>
<td>Data Builder</td>
</tr>
<tr>
<td><strong>Code2seq</strong></td>
<td>We can have the model</td>
</tr>
<tr>
<td><strong>Graph2seq</strong></td>
<td>LSTM</td>
</tr>
<tr>
<td><strong>T5-small</strong></td>
<td>Visualize the images</td>
</tr>
</tbody>
</table>
Plot the model’s performance

We can have the model test set

Plot model performance

Plot the model’s performance

The next step is a lot of the training and test set

Table 8: Example: Plot Model Performance

<table>
<thead>
<tr>
<th>Documentation</th>
<th>Ground Truth</th>
<th>Our Model</th>
<th>Code2seq</th>
<th>Graph2seq</th>
<th>T5-small</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Plot Monthly Mean</td>
<td>Monthly Count</td>
<td>Feature Engineering</td>
<td>Creating a new column</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Example: Count Monthly Mean