Cell2Doc: ML Pipeline for Generating Documentation in Computational Notebooks

Tamal Mondal
IIT Hyderabad
cs21mtech12001@iith.ac.in

Scott Barnett
Deakin University
scott.barnett@deakin.edu.au

Akash Lal
Micorosoft Research
akashl@microsoft.com

Jyothi Vedurada
IIT Hyderabad
jyothiv@cse.iith.ac.in

***Abstract*—Computational notebooks have become the go-to way for solving data-science problems. While they are designed to combine code and documentation, prior work shows that documentation is largely ignored by the developers because of the manual effort. Automated documentation generation can help, but existing techniques fail to capture algorithmic details and developers often end up editing the generated text to provide more explanation and sub-steps.**

This paper proposes a novel machine-learning pipeline, Cell2Doc, for code cell documentation in Python data science notebooks. Our approach works by identifying different *logical contexts* within a code cell, generating documentation for them separately, and finally combining them to arrive at the documentation for the entire code cell. Cell2Doc takes advantage of the capabilities of existing pre-trained language models and improves their efficiency for code cell documentation. We also provide a new benchmark dataset for this task, along with a data-preprocessing pipeline that can be used to create new datasets. We also investigate an appropriate input representation for this task. Our automated evaluation suggests that our best input representation improves the pre-trained model's performance by 2.5x on average. Further, Cell2Doc achieves 1.33x improvement during human evaluation in terms of correctness, informativeness, and readability against the corresponding standalone pre-trained model.

I. Introduction

Computational notebooks enable literate programming [23] by interweaving code, documentation, results, and visualizations into a single file. Notebooks are used by scientists and programmers to communicate insights from data and to create prototypes. Despite their wide adoption, notebooks suffer from poor documentation practices [21], [36], [41]. Notebooks frequently lack explanatory text and one requires access to other information sources such as emails, README files, and presentation slides in order to understand them [36].

Recent advances in deep learning for documentation generation are a promising approach for notebooks to improve their reusability and increase productivity [42]. However, in a study by Wang et al. [31] on these dedicated documentation generation models [29], [42], participants edited the generated text in 41% of cases, often adding more explanation and specific algorithmic details to the machine-generated documentation, sometimes breaking down steps into substeps. Further, training such bespoke models for documenting notebooks is wasteful as pre-trained models are not reused, thus requiring extensive computing and data preparation to replicate.

```
parameters = {
  'application': 'binary', 'objective': 'binary',
  'metric': 'auc', 'is_unbalance': 'true',
  'boosting': 'gbdt', 'num_leaves': 31,
  'feature_fraction': 0.5, 'bagging_fraction': 0.5,
  'bagging_freq': 20, 'learning_rate': 0.05,
  'verbose': 0
}
train_data = lightgbm.Dataset(X_train,
   label=y_train, categorical_feature=cat_cols)
val_data = lightgbm.Dataset(X_val, label=y_val)
model = lightgbm.train(parameters,
   train_data, valid_sets=val_data,
   num_boost_round=5000, early_stopping_rounds=100)
```

Fig. 1. Code cell from a notebook on `kaggle.com`

One solution is to use a pre-trained open-source model that can be fine-tuned for this downstream task. We find (in line with our human study) that existing pre-trained models [6], [11], [14]–[16], [20], [43], even when fine-tuned for documenting notebooks, (1) perform poorly on long code snippets, and (2) suffer information loss due to the structural limitations of their architecture. Consider the code snippet shown in Figure 1. It performs a training task where it first initializes several parameters, loads the training and validation data, and then trains a model. Figure 2 shows the output documentation for this code snippet from CodeT5 [43], PLBART [6], CodeBERT [14], GraphCodeBERT [16] and UniXcoder [15]. The generated text does not capture complete information about the code.

Another solution is to use large language models with code understanding such as GPT-4 [5] to document notebooks. However, these models are closed-source and require extensive resources to operate, preventing wide adoption.

To address these issues, we propose a novel solution called Cell2Doc. Instead of building new models dedicated to the documentation generation task (like [29], [42]), Cell2Doc introduces a new ML pipeline that takes advantage of the capabilities of existing models and enhances it for documentation generation. The results of Cell2Doc, when plugged in with CodeT5, CodeBERT, and UnixCoder, are shown in Figure 2. They capture much more information related to all the tasks in the code snippet compared to directly using the pre-trained models (confirmed by our evaluation).

Cell2Doc is based on the key idea of separating a code

Model	Generated Documentation
CodeT5	Now we will train the lightgbm model
PLBART	Train the model
CodeBERT	Light GBM Model
GraphCodeBERT	Train the model
UniXcoder	Train the model
Cell2Doc (with CodeT5)	STEP 1: Create the parameters for LGBM STEP 2: Create the training and validation data containers STEP 3: Train the model
Cell2Doc (with CodeBERT)	STEP 1: Create the parameters for the model STEP 2: Creating the data structures STEP 3: Train the model with early stopping
Cell2Doc (with UnixCoder)	STEP 1: parameters for LGBM model STEP 2: Light GBM data generator STEP 3: Train the model

Fig. 2. Documentation generated by various models for the code in Figure 1

cell into logically separable segments and documenting them independently to capture multiple operations and algorithmic details in a code cell. Our user study found that 60% of the participants preferred our algorithmic documentation format. Figure 3 shows the working of our Cell2Doc pipeline on the code in Figure 1. The first stage uses a novel *Code Segmentation Model* (CoSEG) to split the input code into logically separable segments. The second stage uses a *Code Documentation Model* (CoDoc) that generates documentation for each code segment separately, and the generated texts are then combined to get the complete documentation.

CoSEG is a classification model that predicts whether a new context starts at a given code line and is designed to address the problem of short document generation that was faced by previous models. CoSEG produces relatively smaller code snippets for the CoDoc stage that helps mitigate the problem of capturing longer code dependencies. It also helps to deal with the risk of input truncation and subsequent information loss when CoDoc only takes a fixed-length input. CoDoc can be any pre-trained model that had been fine-tuned for document generation. For best performance, we created a good-quality dataset using notebooks extracted from the Kaggle repository. For precise ground-truth documentation labels in the dataset, we use text summarization techniques [27], [38], in contrast to previous efforts that relied on heuristics for ground truth; these heuristics fail in many cases on real notebooks.

The key contributions of this work are as follows:

- A novel machine learning pipeline for code cell documentation in Python data-science notebooks that enhances the capabilities of large pre-trained models.[1]
- A novel code segmentation approach that splits input code into multiple logical contexts.
- A dataset (18,378 data points consisting of a pair of Python code and precise documentation) for document generation task in Python notebooks.
- An extensive empirical evaluation of our approach with different input representations. For the code documentation task, our best input representation achieves BLEU scores of 32.82 (with CodeT5), 31.82 (with PLBART), 31.31 (with CodeBERT), 30.7 (with GraphCodeBERT) and 31.97 (with UnixCoder) compared to raw code-markdown pairs that achieve 13.21, 12.5, 12.11, 11.39, and 13.14, respectively.
- A user study evaluating the developer preferences for the generated documentation. The qualitative evaluation shows that the Cell2Doc pipeline achieves significantly better scores in terms of *correctness*, *informativeness*, and *readability* with respect to standalone pre-trained models.

[1]All the codes and data are available at https://github.com/jyothivedurada/KaggleDocGen

II. Background

Existing pre-trained models vary in their architecture, input representation, and pre-training objective. "Encoder-only" models provide embedding representations of input, which are used for code understanding, and "Decoder-only" models decode embedding representations to target sequences. "Encoder-Decoder" models can do both and are typically used for generation tasks like code translation.

CodeBERT [14] is a bimodal pre-trained model that accepts both programming language (PL) and natural language (NL) as input. GraphCodeBERT [16] is similar but takes the data flow of the code into account, emphasizing the semantic structure in the code rather than its syntax. CodeBERT and GraphCodeBERT are encoder-only models, and even though these models work well for code understanding and generation tasks with respect to previous works, they work suboptimally, especially for generation tasks like code summarization because they do not have a pre-trained decoder. Encoder-decoder models like CodeT5 [43], PLBART [6], and UniXCoder [15] are able to support both understanding and generation tasks well because of their unified pre-training.

Fine-tuning involves further training on a task-specific dataset. As the pre-trained models already understand programming language and the natural language, the time and dataset size requirements for fine-tuning are relatively small compared to training from scratch. Depending on the downstream task, pre-trained models can be fine-tuned in multiple ways. For tasks like classification (e.g., bug detection), where one requires the overall representation of the input, the output embedding of `[CLS]` or `<s>` token (a special token added at the beginning of the input code for classification) is used by a neural network for classification. When using encoder-only models (like CodeBERT and GraphCodeBERT) for a generative task like code documentation, the output embedding of each input token can be used by a decoder for generation in a sequence-to-sequence manner. Encoder-decoder models (like CodeT5, PLBART, and UniXcoder) do not need a separate decoder and can be directly fine-tuned for the generative tasks.

III. Dataset for Documentation in Notebooks

To the best of our knowledge, there is no prior open-source code-documentation dataset for Python data-science

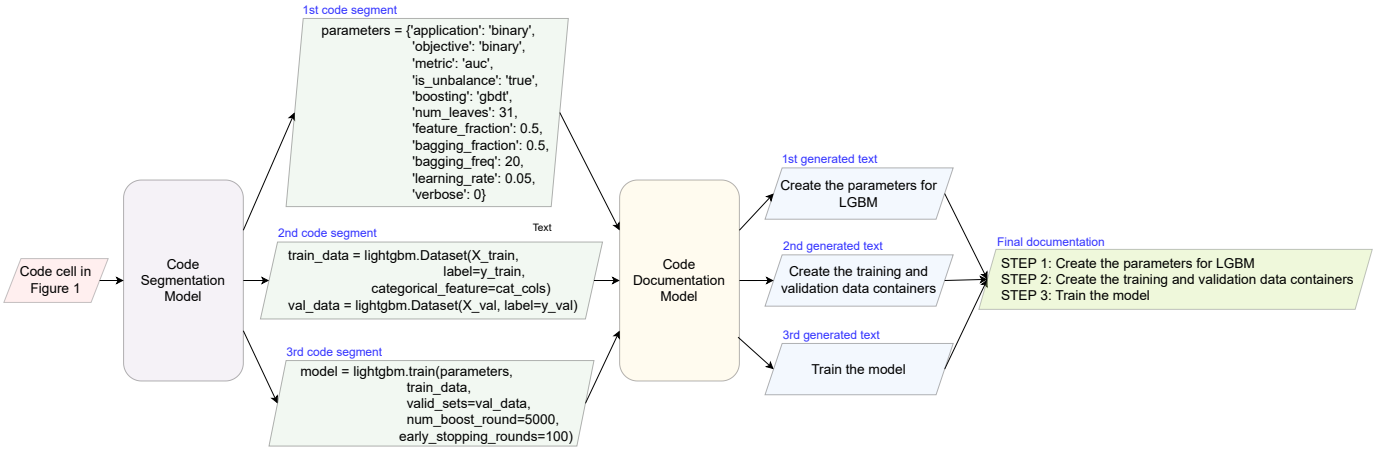

Fig. 3. Example illustrating our Cell2Doc (with CodeT5) pipeline using the code snippet shown in Figure 1

notebooks.[2] We provide a good-quality dataset for this task.

A. *Data Collection*

We collect the notebooks in our dataset from the Kaggle platform. Kaggle is an online community of data scientists and machine-learning practitioners that hosts machine-learning competitions and offers the infrastructure for solving data-science problems. While GitHub is another source, previous studies have suggested that the quality of notebooks on GitHub is poor in terms of code description or the number of markdown cells present in those notebooks [36]. Kaggle users can "upvote" a notebook if they find it useful; ones with a good number of upvotes can be considered well-documented [41].

We collected the raw notebooks from KGTorrent [35] that has a corpus of around 250K notebooks downloaded from Kaggle on 27th Oct, 2020. We applied two filters to this corpus. First, we only considered notebooks with ten or more upvotes. Second, we restricted focus to coding-related notebooks, as opposed to ones intended for tutorials or blogs, by only considering notebooks from Kaggle competitions.

The above filters resulted in 5428 notebooks, from which we extracted code and markdown cells. For every code cell, we consider its immediately previous markdown cell as its documentation. Inline comments in code also serve as documentation, thus, we collected comments and their immediately following code snippet as additional data points. In total, we get roughly 15K and 21K code-markdown and code-comment pairs, respectively. Furthermore, as suggested by previous work [18], [25], we only consider data points where the documentation length was at least three tokens and the code length was at least three lines and at most 100 tokens. Figure 4 summarizes the statistics of our collected dataset.

B. *Dataset Preprocessing*

Markdown cells cannot directly be used as the ground-truth documentation because their contents do more than

[2]The dataset in [29] is not usable because it does not contain actual code-documentation pairs or preprocessed data or preprocessing steps.

Property	Max	Avg	Min	75%
Code tokens for code-markdown pairs	99	53	2	72
Documentation tokens for code-markdown pairs	1131	28	3	32
Code tokens for code-comment pairs	99	52	3	69
Documentation tokens for code-comment pairs	810	10	3	11

Fig. 4. Distribution of code and documentation token lengths

summarizing the following code; they may cover background, theory, reasoning, etc. (see examples in Figure 5). To mitigate this issue, we use NLP-based automatic text summarization solutions to get short and less noisy documentation.

Text summarization is the process of reducing a large piece of text to a shorter one by extracting important information from it. Text summarization techniques can be classified into two types: *Extractive* and *Abstractive*. In extractive summarization, the summarized text contains words and sentences that are part of the original text, whereas, in abstractive summarization, new words and sentences may be added to the summarized text. We tried both kinds of summarization techniques. For extractive summarization, we used Spacy [38], a popular, open-source NLP library used for the production-ready development of NLP tasks. Text summarization using Spacy is based on word frequencies. For abstractive summarization, we used BART [27], which is a transformer-based Seq2Seq model that uses deep learning. (We have directly used the BART checkpoint [17] that was pre-trained and fine-tuned on the CNN Daily Mail dataset for the text summarization task.) Figure 5 shows some examples of Spacy and BART summarization from our dataset. As shown in Example 1 (first row), both Spacy and BART extract important information from the original text and significantly reduce the documentation length. Furthermore, Example 1 shows that taking only the first line (as done in previous work [29]) does not convey the correct information. In the case of Examples 2 and 3, even though the length reduction is not much, summarization still removes the noise in the original documentation. Using both

Ex.	Code Snippet	Original Documentation	Spacy Summarization	BART Summarization

1	def objective(values):
	  params = {
	  'max_depth': values[0],
	  'num_leaves': values[1],
	  'min_child_samples': values[2],
	  'scale_pos_weight': values[3],
	  'subsample': values[4],
	  'colsample_bytree': values[5],
	  'metric':'auc',
	  'nthread': 8,
	  'boosting_type': 'gbdt',
	  'objective': 'binary',
	  'learning_rate':0.15,
	...
		Below is the fun part. The function gp_minimize requires an objective function and what the function all needs is basically a metric we want to minimize. **Of course, we can just use whatever training setup we have been using but just tweak it to return a AUC to minimize..(negative AUC)**	The function gp_minimize requires an objective function and what the function all needs is basically a metric we want to minimize.	The function gp_minimize requires an objective function. What the function all needs is basically a metric we want to minimize.

2	target = train['trip_duration']
	print("Longest trip duration {} or {} minutes:
	  " .format(np.max(target.values), np.max(
	  target.values)//60))
	print("Smallest trip duration {} or {} minutes:
	  ".format(np.min(target.values),np.min(
	  target.values)//60))
	print("Average trip duration : {} or {} minutes
	  ".format(np.mean(target.values), np.mean
	  (target.values)//60))
		No overlap in train and test ids. Hence we can drop the id column for now until unless we figure out that there actually was some overlap**(by some magic of course!)**	Hence we can drop the id column for now until unless we figure out that there actually was some overlap(by some magic of course!)	No overlap in train and test ids. Hence we can drop the id column for now until we figure out that there actually was some overlap.

3	from matplotlib import pyplot
	from keras.preprocessing.image import
	  ImageDataGenerator
	from keras.models import Sequential
	from keras.layers import Conv2D, MaxPooling2D,
	  Dense, Dropout, Input, Flatten,
	  Activation
	...
		Get back to building a CNN using Keras. **Much better frameworks then others. You will enjoy for sure.**	Get back to building a CNN using Keras.	Get back to building a CNN using Keras. Much better frameworks then others.

Fig. 5. Summarization examples using Spacy and BART

Spacy and BART summarized text as ground truth, we got similar accuracy numbers for the code documentation task, and so for all our experiments mentioned in the paper, we used Spacy summarization because it takes significantly less time to process than BART.

Markdown sometimes contains a *header* that describes the rest of the text in 3-5 words. It is also possible to retrieve concise documentation through headers, just like text summarization. Hence, in our dataset creation, we considered only the header as the documentation whenever the header was present. Further, from the documentation, we removed unnecessary parts such as URLs, HTML tags, escape characters, multiple spaces, etc., as part of the preprocessing.

Processing of code is much simpler than processing of documentation: we just removed the single-line and multi-line comments that were present in the code.

We divided the notebooks into training, test, and validation data in an 8:1:1 ratio. We split the dataset at the notebook level; previous studies have suggested that one project should only be included in one set of the split to avoid information leakage [26]. Figure 6 summarizes the statistics of our final processed dataset for the code documentation task. Our preprocessing does not rely on manual effort and is able to significantly reduce the number of tokens in ground-truth documentation (see Figures 4 and 6).

Property	Training	Validation	Test
Number of notebooks	3073	357	377
Code-Documentation pairs	14802	1821	1755
Min. # tokens in Documentation	3	3	3
Avg. # tokens in Documentation	6	6	3
Max. # tokens in Documentation	13	13	13
Min. # tokens in Code	3	5	3
Avg. # tokens in Code	52	51	52
Max. # tokens in Code	99	99	99

Fig. 6. Statistics of code documentation dataset

IV. CELL2DOC

A. *Cell2Doc Pipeline*

Figure 3 illustrates the Cell2Doc pipeline. It accepts a code snippet as input (from a notebook code cell). In the first stage, Cell2Doc splits the input into multiple logical segments using a *Code Segmentation Model* (CoSEG). In the second stage, documentation is generated separately for each segment using a *Code Documentation Model* (CoDoc), and the output is combined in the form of a step-wise algorithm to get the complete documentation. We observed that the multiple segments describe various tasks in a code cell and are suitable to be displayed as independent steps; our human evaluation (in section V-B and VI-0a) further supports this observation.

B. *Code Segmentation Model*

This section explores an important question: *Can an automatic system identify different tasks/operations in a given code snippet?* We show that: with a well-constructed dataset, a machine-learning model can learn this information from the locations of real in-line comments added by developers and from Abstract Syntax Tree (AST) code structures present in notebook code cells. We refer to such a model as the Code Segmentation Model (CoSEG).

*a) **CoSEG Construction**:* We build CoSEG, a binary classification model, by fine-tuning a pre-trained CodeBERT model to determine if the context changes at a particular line (*the line of interest*), given its *prefix* code. We provide the input in the following form: $\texttt{[CLS]}pc_1, pc_2, pc_3, \cdots \texttt{[EOS]}$ $\texttt{[CLS]}c_1, c_2, c_3, \cdots \texttt{[EOS]}$, where [CLS] and [EOS] marks

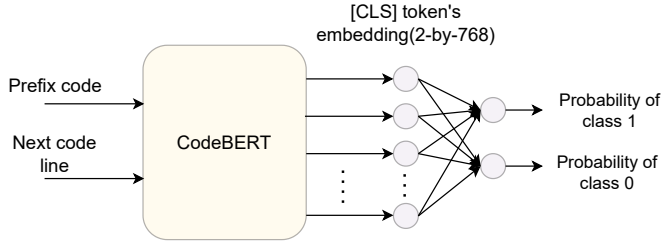

Fig. 7. CodeBERT fine-tuning for code segmentation task

the beginning and the end of a code-snippet, the pc_i represent the tokens of the prefix code and the c_i represent the tokens in the line of interest. These tokens are collectively referred to as $input\_tokens$. The model outputs 1 if a new context begins at the line of interest, 0 otherwise.

Figure 7 shows the fine-tuning of the CodeBERT model for this task. CodeBERT provides embeddings of length 768 for each of the input code tokens as follows:

$$[H_{cls1}, H_{pc1}, \cdots, H_{eos1}, H_{cls2}, H_{c1}, \cdots, H_{eos2}] = CodeBERT(input\_tokens) \quad (1)$$

From the CodeBERT output, we consider the embedding of the two `[CLS]` tokens (H_{cls1} and H_{cls2}) that provide the representation of the prefix code and the line of interest, respectively. To get the combined representation, we merge the two embeddings as: $H = H_{cls1} + H_{cls2}$.

A 2-layer neural network classifies this combined representation by using *softmax* activation to determine the probabilities for each class in the last layer. We compute the binary cross-entropy loss using the predicted probabilities and ground truth labels as follows:

$$Loss = -\frac{1}{M}\sum_{i=1}^{M}(y_i \log(p(y_i)) + (1-y_i)\log(1-p(y_i))) \quad (2)$$

In this equation, M is the number of input data points, y_i is the label for the i^{th} input, and $p(y_i)$ is the predicted probability for the datapoint having label y_i. The loss is back-propagated to CodeBERT and the neural network.

*b) **Segmentation Algorithm**:* CoSEG outputs 1 if a new context begins at the line of interest. Starting from the second line of the input code, the segmentation algorithm, shown in Figure 8, iteratively queries CoSEG on each line to determine whether a code line belongs to the prefix code context or a new context. Note that the last segment is used as the context when querying CoSEG (Line 4).

*c) **Dataset for CoSEG**:* There are no existing labels for code segmentation, so we needed to create one for training. The challenge is to identify the code lines that divide two different contexts for documentation in a code snippet. We considered two criteria for obtaining such code lines. The first criterion is to identify locations of in-line comments added by developers in notebook code cells. An in-line comment typically explains the code below it and separates the code

Input: C: Code as $[c_1, c_2, \ldots, c_n]$, where c_i is a code line; M: Code segmentation model
Output: S: Segmented code as $[s_1, s_2, \ldots, s_m]$, where s_j is a code segment

```
1 Procedure segmentation(C, M)
2     S ← [[c_1]];
3     for i ← 2 to n do
4         prediction ← M(S.last, c_i);
5         if prediction = 1 then
6             S = S ⊕ [[c_i]] ;          // Start a new segment
7         else
8             S.last = S.last ⊕ [c_i] ;  // Add to prev. segment
```

Fig. 8. Code Segmentation Algorithm

segments present above and below it, which can be regarded as splitting two different logical contexts. The second criterion is to identify contexts based on Abstract Syntax Tree (AST) code structures. Different AST structural elements such as definitions (`def`), control-flow statements (`for`/`while`/`if`), classes (`class`), exception handlers (`try`) etc. (compound statements [3] in python) are logical segments that can represent individual contexts for documentation. In our dataset creation process, we first collected these logical contexts (referred to as segments) from notebook code cells and then constructed positive and negative data examples using them.

The algorithm in Figure 9 describes how we create the code segmentation dataset. The procedure `generateDataPoints()` invokes two functions `splitCodeByComment()` and `splitCodeByAST()` to create logical segments using in-line code comments (at Line 16) and AST structures (at Line 28), respectively. To identify code structures, `splitCodeByAST()` recursively traverses the AST of the input code cell using the Python AST parser [34].

After we get the segments using the two criteria on each code snippet, we collect positive and negative data points. Each data point is a triple of the form (*prefix_code*, *the_line_of_interest*, *label*). The first line in each segment creates a positive example with the code from the previous segment as its prefix code (at Line 7). To create a negative example, we randomly select a line from a segment and use the code lines before it in the same segment as its prefix code (at Line 11). We use the same set of 5428 notebooks and like the code documentation dataset, we have split the dataset at the notebook level to create training, test, and validation data. Figure 10 summarizes the statistics for this dataset. With a total of 340K data points, this large dataset allows the CoSEG model to be more accurate (Section V-A).

*d) **Discussion**:* Given the AST structures in a code can be found statically, the question arises whether code segmentation could be algorithmically addressed, eliminating the need for a machine learning model. As described in the Algorithm (Figure 9), the CoSEG learns two (not one) aspects:

Input: C: Code as $[c_1, c_2, \ldots, c_n]$, c_i is a code line
Output: E: Set of data points in the form of (prefix code, line, label)

Procedure `generateDataPoints(`C`)`
  $S' \leftarrow$ `splitCodeByComment(`C`)`;
  $N \leftarrow$ `parse(`C`)`; // Get Abstract Syntax Tree node(s) 'N' from the input code 'C'
  $S'' \leftarrow$ `splitCodeByAST(`N`)`;
  $[s_1, s_2, \cdots, s_n] \leftarrow S' \oplus S''$;
  for $i \leftarrow 2$ **to** n **do**
    $E \leftarrow E \cup \{(s_{i-1},\ s_i.first,\ 1)\}$; // Positive example: (prefix, code line at breakpoint, True label)
  for $i \leftarrow 1$ **to** n **do**
    $[c_1, c_2, \cdots, c_m] \leftarrow s_i$;
    $l \leftarrow$ `random(`$2, m$`)` ; // Get a random line
    $E \leftarrow E \cup \{([c_1, c_2, \cdots, c_{(l-1)}],\ c_l,\ 0)\}$; // Negative: (prefix, line not at breakpoint, False label)

Function `splitCodeByComment(`C`)`
  $S \leftarrow (c_1\ is\ a\ comment)\ ?\ [[]]\ :\ [[c_1]]$; // Initialize S, where C is $[c_1, c_2, \cdots c_n]$
  for $i \leftarrow 2$ **to** n **do**
    if $c_i\ is\ a\ comment$ **then**
      $S \leftarrow S \oplus [[]]$; // Starts a new segment
    else
      $S.last = S.last \oplus [c_i]$; // Add to prev. segment
  return S;

Function `splitCodeByAST(`N`)`
  $S \leftarrow [[]]$;
  $Structs \leftarrow [$"def", "for", "$while$", "if", "try", ..$]$; // Compound statement types
  for $i \leftarrow 1$ **to** n **do**
    if $N_i.stmtType \in Structs$ **then**
      $N' \leftarrow\ child\ nodes\ of\ N_i$;
      $S' \leftarrow$ `splitCodeByAST(`N'`)` ; // Recursively get segments for child nodes
      $S \leftarrow S \oplus S'$;
      $S \leftarrow S \oplus [[]]$; // Start a new segment because N_i is a compound statement
    else
      $S.last \leftarrow S.last \oplus$ `unparse(`N_i`)` ; // Add to the previous segment
  return S;

Fig. 9. Algorithm for code segmentation data creation

(1) AST code structures and (2) locations of in-line comments in real-world notebook code cells. Learning these aspects is necessary because, during testing, input code snippets might lack compound statements or in-line comments, as shown in Figure 11. As our CoSEG model effectively learns breakpoint placement from extensive data, it accurately generates two segments for this code: the first code segment is for result creation, and the second one is to save the result. If we do not have a learned CoSEG model, the user may have to provide breakpoints to create segments concerning different operations, which is not desirable. Further, rather than having two separate systems (AST and comments based), we have combined them into the CoSEG model.

Split	Notebooks	Total data points	Class 1	Class 0
Training set	4309	275106	116511	158595
Validation set	539	34319	14428	19891
Test set	536	31032	12880	18152

Fig. 10. Statistics of the code segmentation dataset

Original Code Cell

```
result = pd.DataFrame(mpred, columns=['target'])
result['id'] = te_ids
result = result.set_index('id')
print('\\n First 10 lines of your 5-fold average prediction:\\n')
print(result.head(10))
sub_file = 'submission_5fold-average-keras-run-01-v1_' + str(score) + '_' + str(now.strftime('%Y-%m-%d-%H-%M')) + '.csv'
print('\\n Writing submission:  %s' % sub_file)
result.to_csv(sub_file, index=True, index_label='id')
```

Segment 1

```
result = pd.DataFrame(mpred, columns=['target'])
result['id'] = te_ids
result = result.set_index('id')
print('\\n First 10 lines of your 5-fold average prediction:\\n')
print(result.head(10))
```

Segment 2

```
sub_file = 'submission_5fold-average-keras-run-01-v1_' + str(score) + '_' + str(now.strftime('%Y-%m-%d-%H-%M')) + '.csv'
print('\\n Writing submission:  %s' % sub_file)
result.to_csv(sub_file, index=True, index_label='id')
```

Fig. 11. Code segmentation example

C. Code Documentation Model

The Cell2Doc pipeline allows the Code Documentation Model (CoDoc) to be any pre-trained model that is fine-tuned for document generation task. We integrate our pipeline with five such models, CodeT5 [43], PLBART [6], CodeBERT [14], GraphCodeBERT [16] and UniXcoder [15], separately. We chose these models because they are recent, open-source, and have different types of architectures with various input representations and pre-training objective functions (see Section II). We fine-tuned these five models using our domain-specific dataset, as described in Section III-B. Note that we utilized the checkpoints of these five models after first fine-tuning them for the code summarization task on the CodeSearchNet [18] dataset (results in better performance since there were more than 250K Python code-comment pairs in CodeSearchNet).

Since code documentation is a generative task, CoDoc relies on an encoder-decoder architecture to generate documentation, where the encoder embeds the input, and the decoder decodes it into the target sequence. As CodeBERT and GraphCodeBERT are encoder-only models, we use a transformer-based

six-layer decoder to construct CoDoc for both models. On the other hand, we have fine-tuned CodeT5, PLBART, and UnixCoder (using encoder-decoder mode) directly on our code documentation dataset for the documentation task, as these models do not require a separate decoder.

D. *Input Representations*

CoDoc accepts code-documentation pairs as input and needs a suitable input representation to generate good-quality documentation. We explored four different input representations for CoDoc to find the best fit in our Cell2Doc pipeline.

First, we consider the basic input representation, where the input code remains in its original form, and the label contains the entire content of the preceding markdown cell (as shown in the second column of Figure 12). We refer to this representation as *Code - Markdown* (CM).

Second, we consider an improved input representation with a summarized version of the markdown cell, as the label and input code remains the same. As illustrated in Section III-B, summarizing markdown cells leads to shorter, meaningful documentation (see the third column of Figure 12). We call this representation as *Code - Summarized Markdown* (CSM).

Third, we consider only code tokens that consist of English characters, not the complete input code. We remove non-letter tokens such as brackets and numbers resulting in smaller inputs (fourth column of Figure 12). These shorter inputs address the fixed-length constraint of pre-trained models but cannot adequately represent structural aspects of the code. The labels are summarized markdown cells. This representation explains how much the naming of different code tokens matters for document generation in Python data-science notebooks rather than the structure of the code. We refer to this case as *English Code tokens - Summarized Markdown* (ECSM).

Fourth, along with considering complete code and summarized markdown as in the CSM case, we consider inline comments as separate data points, where the label is the comment within a code cell (summarized if longer), and the immediate code following the comment is considered input code. For example, in Figure 12, along with the complete code in the code cell, comment lines 3 and 11 also create two additional data points. These smaller input codes increase the dataset size and help the model generate better documentation for smaller code snippets. Moreover, in-lined comments provide explanations of the immediate code, which aids in retrieving good-quality labels. This representation is called *Split Code - Summarized Comment and Markdown* (SCSCM).

V. Evaluation

We evaluate the following research questions for Cell2Doc.
RQ1. How effective is the code segmentation model (CoSEG) in identifying different contexts within a code cell?
RQ2. How do human participants assess Cell2Doc's output?
RQ3. How effective is the code documentation model (CoDoc) for various input representations?
RQ4. How does the Cell2Doc pipeline generalize to different pre-trained models?
RQ5. Can we use the generative (decoder-only) models as CoDoc?

We could not include some of the prior work on notebook documentation model, namely HAConvGNN [29], [42], in our evaluation because although its code is open source, no pre-processing steps are available to run it on our data; the authors of this work have also not responded to our attempts to contact them in this regard.

*a) **Experimental setup**:* We conducted all our quantitative experiments on an Nvidia DGX-2 server having two Intel Xeon Platinum 2.7 GHz CPUs and 16 Nvidia Tesla V100 GPUs, each with 32 GB memory. Our experiments made use of a maximum of four GPUs. We trained CoSEG for two epochs (the training dataset was quite large and there was no significant improvement in classification scores after two epochs), with a learning rate $5e^{-5}$ and batch size 16. We trained CoDoc using learning rate $5e^{-5}$ and batch size eight until there was no improvement in the BLEU score [33] on validation data for three consecutive epochs. We trained both CoSEG and CoDoc models using Adam (ϵ = $1e^{-8}$) optimizer [22]. We implemented the Cell2Doc pipeline in Python 3.6.9 using PyTorch. For all the models, pre-trained weights were taken from Huggingface [4].

*b) **Metrics Used**:* Several studies have examined the challenges of quantitatively evaluating machine-generated text [8], [37]. To help mitigate these challenges we report multiple metrics. First, we report commonly used metrics for evaluating generated text: Bilingual Evaluation Understudy (BLEU) [33] and Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [28]. BLEU calculates a score based on n-gram precision, whereas ROUGE captures n-gram recall. The issue with both approaches is that they focus on token overlap without considering token order or meaning. Finally, we report the BERT score [46] that is calculated as the cosine similarity of the contextual embeddings computed by BERT on the candidate and reference sentences. In the case of the ROUGE score, we report the F1 scores for Rouge-1, Rouge-2, and Rouge-LCS. As we have formulated the code segmentation problem as a binary classification problem, we report precision, recall, and F1 score.

A. RQ1: Effectiveness of the Code Segmentation Model

Figure 13 summarizes the precision, recall, and F1 score of CoSEG. Further, on the test data, we found that the ROC curve for the code segmentation model had an Area Under The Curve (AUC) of **0.964**, indicating its effectiveness.

The classification accuracy numbers, along with the ROC curve, indicate that the classifier is effective in understanding the similarity in context (see Section IV-B) between the prefix-code and a given code-line while making the prediction.

B. RQ2: Human Evaluation of Cell2Doc Output

Automated evaluation of Cell2Doc output is challenging because it generates documentation as algorithmic steps that is different in structure from the ground truth labels. Hence, we use a human evaluation to understand its effectiveness.

Code	Markdown	Summarized Markdown	English Code Tokens

`TRAINING_SIZE = 300000`
`TEST_SIZE = 50000`
`# Load data`
`train = pd.read_csv('../input/train.csv',`
`skiprows=range(1,184903891-TRAINING_SIZE-TEST_SIZE),`
`nrows=TRAINING_SIZE,`
`parse_dates=['click_time'])`
`val = pd.read_csv('../input/train.csv',`
`skiprows=range(1,184903891-TEST_SIZE),`
`nrows=TEST_SIZE,`
`parse_dates=['click_time'])`
`# Split into X and y`
`y_train = train['is_attributed']`
`y_val = val['is_attributed']`
	Instead of doing cross-validation, in this notebook I'm going to use a simple fixed training and testing set. Just for illustration purpose.	Instead of doing cross-validation, in this notebook I'm going to use a simple fixed training and testing set.	['TRAINING_SIZE', 'TEST_SIZE', 'train', 'pd', 'read_csv', '"../input/train.csv"', 'skiprows', 'range', 'TRAINING_SIZE', 'TEST_SIZE', 'nrows', 'TRAINING_SIZE', 'parse_dates', '"click_time"', 'val', 'pd', 'read_csv', '"../input/train.csv"', 'skiprows', 'range', 'TEST_SIZE', 'nrows', 'TEST_SIZE', 'parse_dates', '"click_time"', 'y_train', 'train', '"is_attributed"', 'y_val', 'val', '"is_attributed"']

Fig. 12. Different components of a datapoint

Split	Precision	Recall	F1 score
Training data	87.94	92.82	90.31
Validation data	84.89	88.42	86.62
Testing data	85.88	89.39	87.6

Fig. 13. Precision, Recall and F1 score of Code Segmentation Model

For this experiment, we used a pre-trained CodeT5 model as CoDoc since it is state-of-the-art for code-summarization task (supported by our automatic evaluation results, see Figures 15 and 16).

*a) **Participants**:* The setup of our evaluation is inspired by previous studies [29]. A total of 21 participants took part in our evaluation, including 11 postgraduate, six undergraduate, and four Ph.D. students. All of them used computational notebooks regularly in their day-to-day work and had different levels of data science and machine learning experience.

*b) **Task**:* We randomly sampled 42 code snippets (each is one entire code cell, excluding comments if there are any) from notebooks not used during training. We generated five code documentation texts for each code snippet; four of them are outputs of CodeT5 models trained on data with the four different input representations (Section IV-D), whereas one is an output of Cell2Doc with CodeT5 (fine-tuned on SCSCM) as the CoDoc. When the original ground truth from the notebook is also taken into account, a total of six documentation texts must be assessed per code snippet. (Note: while we get six human ratings for each example, this section presents an assessment of the Cell2Doc pipeline's output, not compare input representations, which is done in the RQ3 section.) We randomly split the 42 samples into seven sets, each with six samples. Three participants evaluated each set independently. The task involved understanding the code on their own and then assessing the quality of the documentation texts. We gave the participants a week to complete the human evaluation at their convenience. Note that the participants did not know which documentation came from what model.

*c) **Dimensions and Scale**:* We used *Correctness*, *Informativeness*, and *Readability* as the dimensions for evaluation, following prior work [29]. Participants rated each code documentation on each of these dimensions with a five-point Likert scale, where one and five indicate the lowest and highest quality, respectively. Correctness measures if the documentation explains the code snippet correctly. Informativeness measures if the documentation captures all the information about the working of the code snippet. Readability measures if the documentation is easy to read and understand.

*d) **Results**:* Figure 14 summarizes the results of the human evaluation. We report the mean and standard deviation of the ratings that the participants gave for the five models and the ground truth across the three dimensions. We can see that Cell2Doc has higher scores than the baselines across all three dimensions. Additionally, while Cell2Doc uses the same code documentation model as CodeT5 (SCSCM), Cell2Doc has better scores than standalone CodeT5, which indicates the benefit of using code segmentation.

The ground truth, surprisingly, did not receive the best score, suggesting that the description was inadequate. By documenting individual contexts separately, Cell2Doc achieves the highest score. Further, we requested the participants for more detailed feedback; we discuss our findings in Section VI.

*e) **Statistical Significance**:* We have used a one-way ANOVA ($\alpha = 0.05$) test to understand if the human evaluation results are significant. For the correctness dimension, ANOVA shows that $F = 16.33 > F_{crit} = 2.22$ with $p \approx 2.55E^{-15} (< 0.05)$, indicating strong statistical significance. Similarly, for informativeness, $F = 22.91$ with $p \approx 9.26E^{-28}$ and for readability, $F = 6.35$ with $p \approx 8.63E^{-06}$. A follow-up Tukey's test ($p < 0.05$) indicates that Cell2Doc's scores significantly differ from those of the other five cases in all three dimensions as it assigns a different group for Cell2Doc. Overall, the results indicate that Cell2Doc has improved the effectiveness of the pre-trained CodeT5 model.

C. RQ3: Evaluation of Code Documentation Model on Different Input Representations

Each row in Figure 15 shows the performance of CodeT5 when fine-tuned on data with CM, CSM, ECSM, and SCSCM representations (from Section IV-D), respectively. Figure 14 (row index 2-5), illustrates how humans rated the output of these four fine-tuned models against sample code examples. Different rows in Figure 15 cannot be directly compared because their test inputs and ground truths differ depending on the input representation, however, since the base model is the same (CodeT5), we can still examine the effectiveness of each fine-tuned model relative to the others.

Index	Model	Correctness	Informativeness	Readability
1	Ground-truth	$\mu = 2.88, \sigma = 1.231$	$\mu = 2.825, \sigma = 1.106$	$\mu = 3.769, \sigma = 1.113$
2	CodeT5 (CM)	$\mu = 2.92, \sigma = 1.165$	$\mu = 2.738, \sigma = 1.113$	$\mu = 3.595, \sigma = 1.099$
3	CodeT5 (CSM)	$\mu = 2.825, \sigma = 1.175$	$\mu = 2.587, \sigma = 1.033$	$\mu = 3.619, \sigma = 1.075$
4	CodeT5 (ECSM)	$\mu = 3.007, \sigma = 1.101$	$\mu =, 2.841\sigma = 0.987$	$\mu = 3.69, \sigma = 1.108$
5	CodeT5 (SCSCM)	$\mu = 3.166, \sigma = 1.219$	$\mu = 2.92, \sigma = 1.116$	$\mu = 3.785, \sigma = 1.131$
6	CodeT5 (Cell2Doc)	$\mu =$ **3.944**$, \sigma = 1.048$	$\mu =$ **4.023**$, \sigma = 1.003$	$\mu =$ **4.253**$, \sigma = 0.89$

Fig. 14. Results of the human evaluation on Cell2Doc output and CoDoc outputs

Input Representation	Evaluation on Test data				
	BLEU	ROUGE-1(F1)	ROUGE-2(F1)	ROUGE-l(F1)	BERT
CM	13.21	22.75	10.89	24.88	10.003
CSM	21.64	27.89	13.12	30.13	18.199
ECSM	18.94	24.57	11.49	26.86	15.807
SCSCM	**32.82**	**38.88**	**25.95**	**40.99**	**31.151**

Fig. 15. Quantitative evaluation of CoDoc (with CodeT5) on various input representations

Fine-tuning on both summarized comments and markdown as labels (SCSCM) got the best human ratings in all three dimensions and is effective with high accuracy in terms of all three metrics (BLEU, ROUGE, and BERT scores). The results indicate that comments act as good-quality labels, and training the model becomes more effective as the dataset size increases.

With text summarization applied to input markdown cells, the code documentation model is effective (rows 3-5 in Figure 15). The human evaluation (Figure 14) shows that CSM has a comparable score to CM, unlike automatic evaluation in which CSM outperforms CM. Although CM usually has longer documentation, it sometimes outputs erroneous documentation, as explained in section IV-C, and we attribute the similarity in scores to the limited number of random code snippets used in the human evaluation.

The performance scores are relatively low when using only English code tokens in input code (ECSM), indicating that addressing the fixed-length constraints of pre-trained models with shorter inputs (at the expense of losing structural information) does not improve performance.

Given these results, we use SCSCM input representation for fine-tuning the CoDoc model used by Cell2Doc.

D. RQ4: Understanding the Generalizability of Cell2Doc

We evaluated the generalizability by replacing the CodeT5 model with other pre-trained models: PLBART [6], CodeBERT [14], UnixCoder [15] and GraphCodeBERT [16]. We fine-tuned UnixCoder on our code documentation dataset using the encoder-decoder mode for the documentation task. In the case of CodeBERT and GraphCodeBERT, we use a six-layer decoder to construct CoDoc (see Section IV-C). As discussed in Section II, CodeBERT and PLBART accept input as raw code tokens, UnixCoder uses AST whereas GraphCodeBERT uses both data-flow graph and code tokens.

Figure 16 shows the automated evaluation results of the four pre-trained models for four of the input representations. Like CodeT5, all these four models are effective with high accuracy on SCSCM. Generated texts of the Cell2Doc pipeline with the four pre-trained models on various sample examples are provided in our GitHub repository [2].

E. RQ5: Using Generative (decoder-only) Models as CoDoc

Traditionally, decoder-only or generative models are effective in generation tasks such as code completion, code refinement, etc., because these models are pre-trained with the Next Token Prediction (NTP) objective. Even though these models by nature are not optimal for tasks like code-to-text translation, recently introduced very large language models (LLMs) like GPT-3 [10], GPT-NeoX [9], GPT-J [12], BLOOM [44], ChatGPT [32], pre-trained on massive and diverse corpora, perform well on this task (that too in a task-agnostic manner). The success of these GPT-style models encouraged us to evaluate them on our notebook dataset to study their suitability to be incorporated with Cell2Doc. The open-source models in this space that are pre-trained with some programming languages are GPT-Neo, GPT-J, BLOOM, etc., whereas models like Codex [11] and ChatGPT are close-sourced.

For evaluation, we have considered BLOOMZ [30], the recently introduced enhanced version of the BLOOM model, which is one of the largest open-source models available in this space. BLOOMZ is available in different sizes, with 560M, 1.1B, 1.7B, 3B, 7B, and 176B parameters. We only experimented with smaller 560M and 1.1B variants due to time and resources (particularly, GPUs) constraints for fine-tuning. Further, we have only fine-tuned these models for two epochs, and that too with a batch size of two for similar reasons. For the prompt during training, we add the "Code:" tag before the code part and the "Documentation:" tag before the ground truth label so the model can understand the structure of the task. During testing, we only give the code with the "Code:" tag in the prompt and keep the "Documentation:" part empty so that the model generates documentation for the given code.

We evaluate the generality of Cell2Doc by replacing the CodeT5 model with BLOOMZ as CoDoc. Figure 17 summarizes the results. We can also query BLOOMZ for individual segments suggested by CoSEG and evaluate the final Cell2Doc output, however, this would require a human evaluation. Note that the BERT score in Figure 17 does have negative scores in a few places and that is due to rescaling done by BERT score library [1] and the conclusion from the results still holds. Like CodeT5, BLOOMZ is also effective with high accuracy

Input Representation	PLBART			CodeBERT			GraphCodeBERT			UnixCoder		
	BLEU	ROUGE-1(F1)	BERT	BLEU	ROUGE-1(F1)	BERT	BLEU	ROUGE-1(F1)	BERT	BLEU	ROUGE-1(F1)	BERT
CM	12.5	21.76	5.733	12.11	19.50	5.015	11.39	17.31	3.109	13.14	20.86	6.940
CSM	19.58	25.92	15.742	19.07	24.26	16.299	18.11	21.83	13.363	21.42	27.21	17.756
ECSM	15.28	20.77	9.327	19.08	23.61	14.678	17.28	20.72	13.046	19.94	25.08	15.134
SCSCM	**31.82**	**37.62**	**29.331**	**31.31**	**36.19**	**26.688**	**30.7**	**34.56**	**26.552**	**31.97**	**36.77**	**28.912**

Fig. 16. Quantitative evaluation of various pre-trained models as CoDoc

Input Representation	BLOOMZ-1.1B			BLOOMZ-560M		
	BLEU	ROUGE-1(F1)	BERT	BLEU	ROUGE-1(F1)	BERT
CM	8.28	17.23	-8.139	3.13	11.12	-29.018
CSM	19.07	26.30	16.084	11.15	18.45	-3.829
ECSM	19.96	26.86	16.929	5.31	10.96	-26.064
SCSCM	**27.06**	**33.38**	**24.537**	**20.36**	**27.87**	**6.720**

Fig. 17. Quantitative evaluation of BLOOMZ model as CoDoc

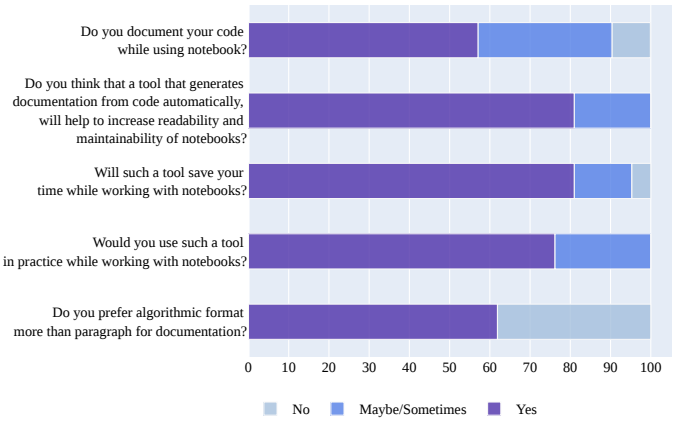

Fig. 18. Post-evaluation feedback

on SCSCM across all metrics. We believe that the relatively lower numbers for BLOOMZ with respect to other models are due to the smaller number of epochs during fine-tuning.

VI. Discussion

a) Post-evaluation Interaction: As part of the human evaluation, we asked the participants for their observations and we now present their feedback (as-is).

The results of the post-evaluation questionnaire, summarized in Figure 18, reveal that over 80% of the participants believe that an automatic documentation tool would increase their productivity, and over 75% expressed their interest in using the tool in practice. Additionally, more than 60% of participants preferred the algorithmic format over the paragraph format for documentation. When asked for the reason behind their preference, P8 mentioned that "A step-wise description is faster and easier to understand than a paragraph explaining the same thing. Additionally, algorithmic steps follow the logic of code, unlike an explanation in a paragraph format." Another participant, P16, commented that "Documentation about what code is meant for or its purpose can be in paragraphs while the steps in implementation are better explained algorithmically."

Few participants explicitly mentioned that the more extended and step-wise documentation from the Cell2Doc pipeline is better than the baselines. For example, P10 said: "The documentation that was in the form of steps, was explaining the code properly in most cases", and P11 said: "The baselines are very weak and not competitive".

According to P3, "The code snippets are not complete so it is difficult to understand sometime without knowing the previous code or complete context". This comment reveals the difficulty of documenting a single code cell without knowledge of previous code cells (even for a human).

P13 mentioned, "If even one step in the generated documentation is wrong it breaks the flow and degrades the overall quality of the documentation". This comment emphasizes the importance of getting each step correct in Cell2Doc to generate efficient combined documentation.

P13 further mentioned that "Documenting each and every step in the source code is not always necessary as some steps are obvious and everyone knows it" which indicates how we are splitting the source code before documenting is very crucial and producing more splits than needed and generating longer documentation might not always help.

b) Threats to Validity: We use automatic text summarization and usual cleaning steps to remove stories and other irrelevant information from the ground truth documentation. Even though we manually analyzed samples and found that, in most cases, it creates relatively better ground truth, we did not formally evaluate it. There are examples where the data cleaning process does not remove the noise or only partially removes the noise.

We test the effectiveness of the Cell2Doc pipeline via human evaluation, which only qualitatively analyzes the generated outputs. It is possible that another quantitative analysis will provide slightly different accuracy numbers. We mitigate this problem by selecting samples randomly for human evaluation and having three participants evaluate each example.

We only used notebooks from the Kaggle platform, which we considered as well-documented notebooks related to data science. There might be other factors to consider when generating documentation for notebooks from other platforms or for uses other than data science.

c) Limitations: The code segmentation model splits code depending on how comments or compound statements appear in the code. This approach may lead to suboptimal contexts since the code segmentation model does not take into account the quality of the generated documentation when learning the contexts (as expressed by one human evaluator P13).

VII. Related Work

a) Code Documentation for Python Notebooks: There have been recent efforts to summarize computational notebooks automatically. HeaderGen [40] generates the header (typically of 1-3 words in length) for a given code cell in Jupyter notebooks by statically analyzing the library calls made in the code. Because of their static mapping of different Machine Learning libraries (such as `matplotlib.pyplot`) to phases in a Machine Learning workflow (such as Visualization), they cannot generate a header for any unknown library without manually updating the mapping.

Themisto [42] is a deep-learning based automatic documentation generation tool for notebooks, providing users with multiple types of documentation, including one based on deep learning, another focused on queries, and one that allows users to enter text. Its human evaluation reports that the tool made it easier to document the notebooks, but in over 50% cases, users created or edited the documentation themselves.

HAConvGNN [29] proposes that documentation relies on more than one code cell. This work employs a combination of Convolutional Graph Neural Network (CGNN) with GRU to encode the AST representation of code and utilizes a hierarchical attention mechanism to combine the representation of four code cells to generate documentation. The human study by Wang et al. [31] included HAConvGNN and suggested that in 41% of the cases, participants added more details to the generated documentation, sometimes explaining steps into substeps.

Themisto comes closest to our work because it generates documentation for a single code cell. Even though HAConvGNN addresses the same problem, unlike us, it considers multiple code cells at once. According to our observations, each code cell contains more information than the current models can extract, and Cell2Doc aims to extract all that information using segmentation in order to generate longer and more meaningful documentation. Furthermore, Cell2Doc enhances the performance of existing pre-trained models to generate effective code documentation.

b) Code Documentation: Several deep learning approaches generate plain language descriptions of code snippets (not specific to notebooks). CODE-NN [19] uses Long Short Term Memory (LSTM) based neural network with an *attention* mechanism to generate documentation for C# code and SQL queries. Graph2Seq [45] uses a Seq2Seq architecture where a graph-based encoder first creates a node and graph embedding, and then a sequence decoder uses this as input to generate the sequence using attention. Code2Seq [7] uses a bag-of-paths approach in which multiple AST paths are considered whose representations are combined using attention to construct the overall representation of the code. [24] also uses the AST representation of the code, but rather than flattening the AST or using some paths of the AST, they use Convolutional Graph Neural Network (ConvGNN) to encode the AST along with attention to learn important tokens from the AST. To perform well in code documentation task, these models need to be trained from scratch, which requires more training time, good infrastructure, and a larger dataset. Transformer-based pre-trained models [39] have provided state-of-the-art results with very limited infrastructure and training, hence, we reuse such models in our Cell2Doc pipeline to efficiently generate documentation.

c) Pre-trained Models for Programming Languages: Pre-trained models like GPT, BERT, and RoBERTa have revolutionized NLP by achieving state-of-the-art results in several NL tasks. The pre-trained models leverage the idea of transfer learning, where the models are initially trained on a large corpus in a self-supervised fashion and later fine-tuned for specific tasks. A number of pre-trained models [6], [11], [13]–[16], [20], [43] have recently been introduced for programming language understanding and generation, similar to the NLP domain and have performed well on tasks like code-to-code translation, clone detection, code generation, etc., with very little additional training.

When using pre-trained models such as CodeT5 [43], PLBART [6], CodeBERT [14], GraphCodeBERT [16] and UniXcoder [15] directly for code documentation, the generated descriptions are generally high-level and less informative in nature. Cell2Doc reuses these models and improves their effectiveness for documentation generation in the case of computational notebooks.

Recently introduced GPT-style decoder-only language models such as GPT-3 [10], Codex [11], GPT-NeoX [9], GPT-J [12], BLOOM [44], ChatGPT [32], GPT-4 [5] are pre-trained on a massive corpus and have a very large number of parameters. These pre-trained models also can be integrated into our Cell2Doc pipeline as a CoDoc model and queried with individual segments suggested by CoSEG. Since many of these generative models are closed-source and consume extensive resources to operate, we evaluated smaller versions of BLOOMZ [30] model, an enhanced version of BLOOM (one of the largest open-source models available).

VIII. Conclusion

Computational notebooks, while popular, suffer from lack of documentation that makes them difficult to reuse. Existing techniques for automated code documentation produce output with low information content, particularly as the input code gets longer. Cell2Doc presents a significant enhancement in this space by effectively identifying different contexts in the input code. This paper presents both a quantitative and a qualitative evaluation of Cell2Doc, supporting its effectiveness. Additionally, we have integrated multiple pre-trained models with Cell2Doc to show that it can improve the effectiveness of existing models as well.

Acknowledgment

We are grateful to all the participants who contributed to our human evaluation. Thanks to the authors of pre-trained models for their open-source and well-documented work. Thanks to all the anonymous reviewers for their suggestions and feedback. Tamal Mondal is supported by a Microsoft Research India grant for his post-graduation studies.

REFERENCES

[1] "Bertscore rescaling," https://github.com/Tiiiger/bert_score/blob/master/journal/rescale_baseline.md.

[2] "Cell2Doc Generated Outputs," https://github.com/jyothivedurada/KaggleDocGen/blob/main/Cell2Doc-Outputs-on-Sample-Examples.pdf.

[3] "Compound statements in Python 3," https://docs.python.org/3/reference/compound_stmts.html, 2009.

[4] "Transformers: State-of-the-Art Natural Language Processing," https://github.com/huggingface/transformers, 2020-10.

[5] "GPT-4 Technical Report," https://cdn.openai.com/papers/gpt-4.pdf, 2023.

[6] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[8] F. Alva-Manchego, C. Scarton, and L. Specia, "The (Un)Suitability of Automatic Evaluation Metrics for Text Simplification," *Computational Linguistics*, vol. 47, no. 4, pp. 861–889, 12 2021. [Online]. Available: https://doi.org/10.1162/coli_a_00418

[9] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, "Gpt-neox-20b: An open-source autoregressive language model," 2022.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[12] EleutherAI, "GPT-J in Huggingface," https://huggingface.co/EleutherAI/gpt-j-6b, 2023.

[13] A. Elnaggar, W. Ding, L. Jones, T. Gibbs, T. Feher, C. Angerer, S. Severini, F. Matthes, and B. Rost, "Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing," 2021. [Online]. Available: https://arxiv.org/abs/2104.02443

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[15] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," 2022. [Online]. Available: https://arxiv.org/abs/2203.03850

[16] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[17] Huggingface, "BART model from Huggingface," https://huggingface.co/facebook/bart-large-cnn, 2020-10.

[18] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019. [Online]. Available: https://arxiv.org/abs/1909.09436

[19] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://aclanthology.org/P16-1195

[20] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20. JMLR.org, 2020.

[21] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3173574.3173748

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[23] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 01 1984. [Online]. Available: https://doi.org/10.1093/comjnl/27.2.97

[24] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–195. [Online]. Available: https://doi.org/10.1145/3387904.3389268

[25] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 795–806. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00087

[26] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," 2019. [Online]. Available: https://arxiv.org/abs/1904.02660

[27] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019. [Online]. Available: https://arxiv.org/abs/1910.13461

[28] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[29] X. Liu, D. Wang, A. Wang, Y. Hou, and L. Wu, "HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks," in *Findings of the Association for Computational Linguistics: EMNLP 2021*. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 4473–4485. [Online]. Available: https://aclanthology.org/2021.findings-emnlp.381

[30] N. Muennighoff, T. Wang, L. Sutawika, A. Roberts, S. Biderman, T. L. Scao, M. S. Bari, S. Shen, Z.-X. Yong, H. Schoelkopf, X. Tang, D. Radev, A. F. Aji, K. Almubarak, S. Albanie, Z. Alyafeai, A. Webson, E. Raff, and C. Raffel, "Crosslingual generalization through multitask finetuning," 2022.

[31] M. J. Muller, A. Y. Wang, S. I. Ross, J. D. Weisz, M. Agarwal, K. Talamadupula, S. Houde, F. Martinez, J. T. Richards, J. Drozdal, X. Liu, D. Piorkowski, and D. Wang, "How data scientists improve generated code documentation in jupyter notebooks," in *IUI Workshops*, 2021.

[32] OpenAI, "ChatGPT," https://openai.com/blog/chatgpt, 2022.

[33] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[34] Python, "AST module in Python 3," https://docs.python.org/3/library/ast.html, 2009.

[35] L. Quaranta, F. Calefato, and F. Lanubile, "KGTorrent: A dataset of python jupyter notebooks from kaggle," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2021. [Online]. Available: https://doi.org/10.1109%2Fmsr52588.2021.00072

[36] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference*

on Human Factors in Computing Systems, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3173606

[37] A. B. Sai, A. K. Mohankumar, and M. M. Khapra, "A survey of evaluation metrics used for nlg systems," *ACM Comput. Surv.*, vol. 55, no. 2, jan 2022. [Online]. Available: https://doi.org/10.1145/3485766

[38] Spacy, "Spacy library," https://spacy.io/, 2017.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[40] A. P. S. Venkatesh and E. Bodden, "Automated cell header generator for jupyter notebooks," in *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*, ser. AISTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 17–20. [Online]. Available: https://doi.org/10.1145/3464968.3468410

[41] A. Y. Wang, D. Wang, J. Drozdal, X. Liu, S. Park, S. Oney, and C. Brooks, "What makes a well-documented notebook? a case study of data scientists' documentation practices in kaggle," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411763.3451617

[42] A. Y. Wang, D. Wang, J. Drozdal, M. Muller, S. Park, J. D. Weisz, X. Liu, L. Wu, and C. Dugan, "Documentation matters: Human-centered ai system to assist data science code documentation in computational notebooks," *ACM Trans. Comput.-Hum. Interact.*, vol. 29, no. 2, jan 2022. [Online]. Available: https://doi.org/10.1145/3489465

[43] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[44] B. Workshop, :, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, and M. G. et al., "Bloom: A 176b-parameter open-access multilingual language model," 2023.

[45] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," 2018. [Online]. Available: https://arxiv.org/abs/1804.00823

[46] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," 2019. [Online]. Available: https://arxiv.org/abs/1904.09675