NBQA: A Dataset for Retrieval and Rewriting of Code for Jupyter Notebooks

Aditya Veerubhotla
Code4Thought
adityasv

Atharva Naik
Code4Thought
arnaik

Yash Butala
Code4Thought
ypb

Abstract

With the recent popularity of code retrieval and generation tasks in the NLP community, several datasets have been proposed for these tasks. However most of these datasets either focus only on retrieval or generation from natural language intents. While some of them do add additional context like previous cells of a Jupyter notebook or attempt to combine retrieval and generation, none of them combine long context, open domain retrieval, and generation in one setting. To bridge this gap, we propose NBQA, a dataset that combines the long context of Jupyter Notebooks, with a large heterogeneous knowledge base in an open-retrieval setting. Through our experiments we show that our dataset has the potential to improve code generators. Link to the GitHub repo: https://github.com/atharva-naik/jupyter_nb_qa.

1 Introduction and Motivation

While the tasks of semantic code search using natural language (Husain et al., 2019; Lu et al., 2021; Huang et al., 2021) and code generation from intents specified in natural language (Yin et al., 2018; Iyer et al., 2018; Agashe et al., 2019; Husain et al., 2019) have been widely explored by the research community, both of these settings deal with isolated intents and relatively smaller blocks of code (program snippets (Yin et al., 2018) or isolated functions (Husain et al., 2019)). A more practical setting is to extend an existing codebase, which however is not performed by practical code completion systems such as Github Copilot [1] which is powered by OpenAI's Codex (Chen et al., 2021a).

To bridge the gap between the needs of a programmer and current research, we propose the a new dataset that would aid in the development of systems that can perform both retrieval and generation. Specifically, our task requires a model to perform retrieval over a knowledge base containing code snippets and API documentation to generate the next cell of a Jupyter notebook. The overall pipeline of the data collection and inference is shown in Figure 1

Our dataset and task setting addresses the following limitations in code search and code generation/completion methods:

• **Personalization in code search:** Current code search datasets focus on retrieving code snippets based on natural language intents (Husain et al., 2019; Huang et al., 2021; Lu et al., 2021; Yin et al., 2018), but the retrieved code may not fully address a developer's information needs and require additional editing to adapt to their current code base. Our dataset aims to address this issue by pairing the most relevant retrieval results with the ground-truth continuation cell as supervision for personalizing the solution.

• **Retrieval Enhanced Machine Learning (REML):** (Zamani et al., 2022) proposed a general architecture for ML models which requires querying, retrieval and response utilization. They show that core principles of indexing, representation, retrieval, and ranking can substantially improve model generalization, scalability, robustness, and interpretability. REML decouples memorization from generalization making the models more parameter efficient. Our dataset aims to achieve these benefits in the domain of code question-answering by means of a open-domain KB.

Improvement over existing CodeQA datasets: Currently, the code generation datasets are either framed as search/relevance detection (Huang et al., 2021; Yao et al., 2018) or source code comprehension (Liu and Wan, 2021). (Huang et al., 2021; Yao et al., 2018) frame CodeQA as a binary classification task where given a query and potential code answer the system needs to classify whether it answers the query. These datasets are insufficient for the training and evaluation of a pipeline that

[1] https://github.com/features/copilot

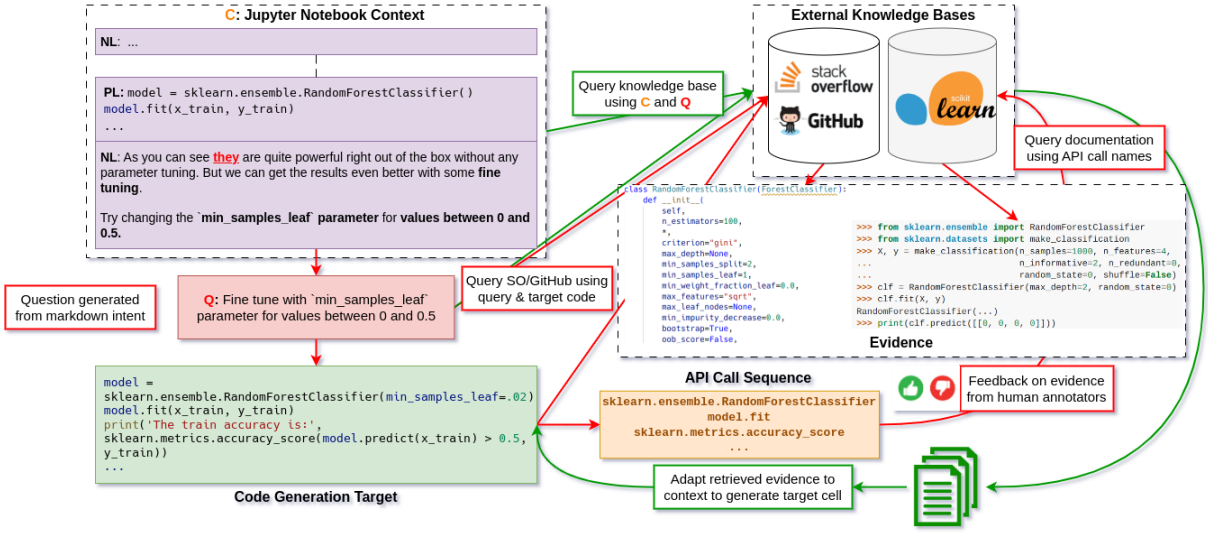

Figure 1: The red arrows show the steps of the data curation pipeline, while the green arrows show the steps of the expected inference pipeline. Data curation: 1) Generate question from markdown intent 2) Gather documentation using API calls from target code 3) Gather class specifications and code usage examples from sources like StackOverflow and GitHub 4) Human feedback on relevance of evidence. Inference: 1) Use query and context to retrieve evidence from external knowledge and documentation 2) Generate target code from gathered evidence

can find the most relevant code for a given query and understand an existing code base well enough to adapt the most relevant solutions to it.

Goals: Our contributions are as follows:
1) Proposed a pipeline for curating KB and Jupyter notebook dataset.
2) Created the dataset for retrieve-then-generate open-domain QA systems for Jupyter notebooks with 160k train instances and 2k curated val+test instances.
3) Show the effectiveness of our dataset at improving code search & code generation systems

2 Related Works

Code Search: (Husain et al., 2019) propose CodeSearchNet, a large-scale corpus for semantic code search with function documentation pairs for 6 programming languages, with 500k function-doc pairs for Python. The CodeXGLUE benchmark (Lu et al., 2021) adds the AdvTest and WebQueryTest test sets derived from CodeSearchNet and CosQA (Huang et al., 2021) respectively. Additionally (Mou et al., 2016) and (Nafi et al., 2019) explore code-to-code retrieval where source code is used as a query to retrieve similar code implementations.

Code QA: CodeQA settings broadly fall under relevance classification (Huang et al., 2021; Kanade et al., 2020; Yao et al., 2018) or source comprehension (Liu and Wan, 2021). The first setting is a binary classification of whether a given code answers the information need or intent of an NL question while the second setting asks NL questions that require source code understanding to answer and expects an NL response. Additional settings like conversational CodeQA for intent clarification in code generation have also been explored by approaches like (Li et al., 2022).

Code Generation: (Yin et al., 2018) create CoNaLa a dataset of paired natural language intent and code sourced from StackOverflow for Python and Java, with 600k automatically mined pairs and 2.8k human annotated pairs with re-written intents for Python. (Yin et al., 2022; Agashe et al., 2019; Huang et al., 2022) tackle the problem of code generation for Jupyter notebooks that involve complex heterogeneous contexts of interleaved code and markdown cells. (Zhou et al., 2022) improves code generation (as measured by exact match and execution) by augmenting prompts given to language models with API documentation. (Zhang et al., 2023) proposed a benchmark RepoEval that is aimed at repository level code completion. It contains repositories covering line, API invocation, and function body completion scenarios and leverages and iterative retrieve then generate approach for code completion.

Question Generation and Rewriting: Question generation is a task of generating valid and fluent natural language questions with given context and answer. Owing to the use of generating questions in

settings such as efficient query writing, chatbots, tutoring systems, many datasets and question generation pipelines (Lee et al., 2020), (Yang et al., 2018), (Chen et al., 2020) have been proposed. (Vakulenko et al., 2020) proposes a query reformulating subtask to modify a conversation into unambiguous questions that can be interpreted outside the context of conversation. (Zhang et al.) works on rewriting tail query into a head query with similar linguistic characteristics as head queries and preserve the shopping intent. In our approach, we focus on writing NL queries based on the intents of the corresponding markdown and target code cell.

Retrieval Augmented Generation: Retrieval augmented generation (RAG) models use external sources to enhance the model memory (Lewis et al., 2020). Due to the non parametric memory, they are better than seq2seq models in generating diverse and accurate texts. Previous works on QA tasks (Izacard and Grave, 2020) and code generation (Yin et al., 2018; Li et al., 2023; Zhang et al., 2023) use retrieval, but with limited or closed context. DocPrompting uses open-domain code generation, but with simple context and small KB (Zhou et al., 2022). We extend this to Jupyter notebook QA with larger context and KB of code snippets and documentations.

3 Datasets

We broadly divide the data sources used in our work into three categories:

Jupyter Notebooks: Jupyter notebooks are self-contained code bases with alternating code and markdown cells, where markdown often explains what the code does. Each instance of our dataset would be a Jupyter notebook context (C), a query to generate a new cell (Q), a target cell to be generated (A), and useful evidences from the knowledge base (E). For the first 3 components we plan to use JuICe, ExeDS & ARCADE (Agashe et al., 2019; Huang et al., 2022; Yin et al., 2022) datasets. The JuICe dataset is a large-scale collection of 1.5M noisy notebooks (having truncated contexts) and nearly 3k curated assignment notebooks, while ExeDS and ARCADE have nearly 500 and 1k notebooks respectively with cell-level execution output to support execution-based evaluation.

Code Examples for Knowledge Base: For collating code examples we used StackOverflow data through StaQC (Yao et al., 2018), CoNaLa (Yin et al., 2018), and GitHub corpora through CodeSearchNet (Husain et al., 2019), PY150 (Raychev et al., 2016a). Additionally, we indexed the classes and function names present in the GitHub repositories of top imported libraries in our corpora.

Documentation Sources: We analyzed the import statements in the JuICe validation set (top-10 packages are shown in Figure 4). The results show that data science is the main focus of the dataset. Similar to (Zhou et al., 2022), we used Devdocs [2] corpus that contains APIs and their usage for multiple libraries in Python. We augment the Devdocs dataset by scarping other libraries which are frequent in the sampled dataset but missing in the Devdocs such as SciPy, PySpark, Seaborn, etc. We use the documentation segmented by <p> HTML tags of the scraped websites.

Deduplication and merging of notebooks: The JuICe dataset suffers from issues of duplication of Jupyter notebook contexts with nearly identical notebooks in some cases for the validation (JVal) and tests splits (JTest), while the training dataset (JTrain) is made by splitting notebooks into multiple context+target cell instances that have overlapping context. We perform deduplication of the JVal and JTest that reduces the numbers of instances from 1830 to 587 and 2114 to 624 respectively and merge the split notebooks in JTrain to get back the original notebooks which reduces the number of instances in JTrain from 1.52M to 263K.

Extending val+test data with JTrain: Due to diversity issues in JVal and JTest stemming from them being sourced from assignments as well as the relatively smaller number of instances, we do a stratified sampling (over markdown to code ratios and complexity of markdowns) of notebooks from JTrain to solve the diversity issues in our val+test data. We call this sampled data JTrainS.

4 Methodology

One of our main contributions is a proof of concept pipeline for re-writing markdown intents (I) as queries (Q) and linking relevant evidence (E) to Jupyter notebook context (C), Query (Q), and target cells/answers (A). Our pipeline scales up this process by automating these steps and requires human intervention/annotation only for some noise reduction (specifically for the validation and test data). We divide our pipeline into four steps including the human annotation:

• **Query rewriting for markdown intents:** We

[2] https://devdocs.io

use the JuICe dataset containing high-quality Jupyter Notebook and contain context (C), Intent (I) and answer (A) triples. However, the intents (I) are often verbose and contain irrelevant information or observations (see table 8). These intents can mislead retrievers by obscuring the user's information need. Therefore, we propose a method to prompt LLMs to generate more compact queries that only include the user's information need by using the Keyphrases in I and API phrases in A. We describe the keyphrase and API phrase extraction steps and the prompting format and LLMs below.

Keyphrase extraction: We use the keyphrases as an approximation of the most important content in markdown intents. We use an ensemble of Keyphrase Boundary Infilling and Replacement (KBIR) (Kulkarni et al., 2022), an state-of-the-art supervised deep-learning based approach trained on the Inspec dataset (Hulth, 2003), and Yet Another Keyphrase Extractor (YAKE) [3] an unsupervised ranking based approach. While KBIR exhibits high precision in extracting Keyphrases, it suffers from a limitation where it outputs no keyphrases for some markdowns (25% of markdowns in JuICe). To overcome this limitation we ensemble it with the YAKE extractor as a fallback model when KBIR fails to extract keyphrases. YAKE is a corpus and language-agnostic approach that works by extracting statistical text features from single documents.

API phrase extraction: We use Abstract Syntax Trees to extract function calls from A using the ast[4] module in Python, subsequently normalizing it by lowering, splitting by underscores, and camel case to obtain phrases from the API calls.

Prompting format: We do the prompting through an instruction and three few-shot examples (as shown in fig 3. Each example contains the processed markdown, keyphrases, and API phrases extracted from the target cell as additional scaffolding for generating the queries. We explicitly instruct the model to include keyphrases and API phrases in the generated query.

LLMs used for prompting: We use GPT-3.5 (Ouyang et al., 2022) (`text-davinci-003`) and LLaMA-65B (Touvron et al., 2023) for our experiments. We use GPT-3.5 for val+test data but use LLaMA for the 60K train instances as GPT is more expensive to run. We also generate queries using both LLMs for a subset of val+test data for a qualitative comparison of the two approaches.

[3] https://github.com/LIAAD/yake

[4] `https://docs.python.org/3/library/ast.html`

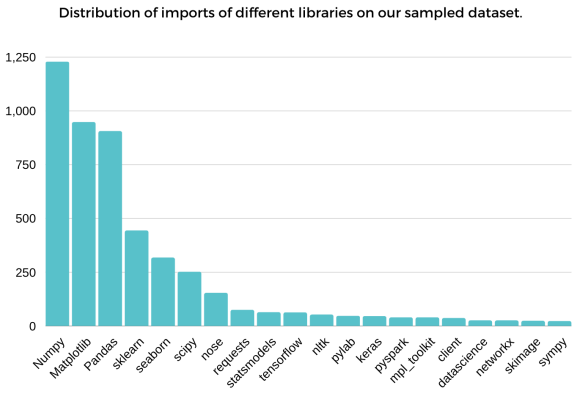

Figure 2: The chart shows the top libraries present in the sampled dataset.

• **KB curation:** The KB consists of two types of data, ie: documentation and code snippets. Our approach to collecting them is as follows:

Documentation: (Zhou et al., 2022) showed that documentations can help the code generation models to generalize to unseen functions and libraries. Hence we created a pool of documentation and aim for a maximum coverage of API functions over our sampled examples. We use the Devdocs dataset that contains a pool of 35763 manuals. They contain documentations of most used Python libraries segmented using <p> HTML tags. We follow the same guidelines to scrape more libraries that make up the top imports but are missing. We exclude the system libraries and standard modules in Python. The pie-chart 2 below shows the most common libraries used in the sampled dataset. Finally, we have a documentation pool of 43972 documents.

Code Snippets: We created an index of different code snippets that models can read from. These snippets have different usages of API calls and also algorithmic or temporal information. We include the Py150 (Raychev et al., 2016b), CodeSearchNet (Husain et al., 2019), along with GitHub repositories of popular libraries which are all sourced from GitHub. StaQC (Yao et al., 2018) and CoNaLa (Yin et al., 2018) which are sourced from Stackoverflow are also included in the KB. We merge the code snippets and index them to keep the code snippets uniform. Finally we have a pool of more than 1.7 million code snippets in our KB.

• **Evidence Retrieval:** We use the evidence retrieval pipeline to collect a set of "high recall" evi-

dence in the form of documentation and code snippets relevant for a notebook context (C), target cell (A), and question (Q). To improve the precision of the evidence we perform binary relevance annotations of the pool of candidate evidences. We also compare the various retriever techniques used using some automated metrics. We used the following retrievers:

Text to code: For text-to-code retrieval for a given instance we use the re-written query Q to fetch codes from our KB that satisfy the intent of Q. We train a dense transformer-based model in a bi-encoder contrastive learning setup (proposed by (Husain et al., 2019)) over the CoNaLa (Yin et al., 2018) and CodeSearchNet (Husain et al., 2019) dataset. For the document/code encoder, we pick SOTA transformer models like CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and UniXcoder (Guo et al., 2022) which have been pre-trained over multimodal code+text data. We also use CodeBERT-python[5] a CodeBERT model that has been pre-trained on additional Python data from the codeparrot dataset[6]. We ensemble the predictions of these approaches by normalizing the relevance scores and re-ranking the union of the predictions.

Code to code: For the code-to-code retrieval pipeline the gold answer/target cell A is treated as the query to fetch similar code examples from our KB. We try a combination of dense and sparse retrievers. For the dense retriever setup we reuse the code/document encoder. We use the code-encoder for representing both the query (A in this case) and the candidate code snippet. For the sparse retriever, we propose CodeBM25 (see appendix F).

Text to text: For text-to-text retrieval we use the re-written query Q as the query and match it with the content of documents (i.e concatenating the title and code/text) in our KB. We ensemble the BM25 sparse, lexical retrieval system with TAS-B (Hofstätter et al., 2021) which is a dense passage retrieval model that leverages dense, continuous, latent representations of the query and document for matching.

API lookup: For API lookup we use static analysis to extract function calls in the target cell A (as shown in figure 1) and then use fuzzy matching to link them to devdocs (and libraries we scraped) documentation for that function call.

• **Human Annotation:** We evaluate the query rewriting and evidence retrieval part of the pipelines through a few pilot annotation studies. We carry out the following studies for it:

$I \rightarrow Q$ **(Faithfulness of Query Q to original markdown Intent I):** We annotate whether the query Q entails intent I (marked as 1). If Q introduces any new information (other than API calls) (neutral) or contradicts any information from I, we mark it as 0.

$Q \leftrightarrow A$ **(Relevance of Question Q to Target Cell A):** We do binary annotations of whether the target cell A satisfies the intent of Q.

$Q/A \leftrightarrow E$ **(Relevance of Evidence E to Query Q and Answer A):** Binary annotation of whether evidence E is relevant to answering Q or could have been used to generate A.

5 Hypothesis and Baselines

We formulate hypotheses to test our assumptions about the helpfulness of re-written queries and retrieved evidence for target cell generation in Jupyter notebooks. We also test a few simple zero-shot baselines on our data to test the support for each hypothesis. For both hypotheses, we track changes in metrics of interest ($\mathcal{F}$), namely CodeBLEU (Ren et al., 2020) score and API-call overlap between the generated code cell $\hat{A}$ and the target cell A.

• **H1: Rewritten queries are better at conveying the intent of target cells:** We hypothesize that the re-writing queries (Q) with API call phrases better elicit the target cell in the Jupyter notebooks when compared to the original markdowns (I). If the model used to generate the code cell $\hat{A}$ is represented by f_θ, we hypothesize that $\mathcal{F}(f_\theta(C, Q)) > \mathcal{F}(f_\theta(C, I))$ where C represents the Jupyter notebook cells that occur before I. It should be noted that queries Q aren't intended to make the task easier, but rather reduce the ambiguity around the target cell.

• **H2: Retrieval augmentation helps cod generation:** We hypothesize that relevant evidences helps in generating the target cell in the Jupyter notebooks. Combined with H1, we can express H2 as $\mathcal{F}(f_\theta(C, Q, E)) > \mathcal{F}(f_\theta(C, Q)) > \mathcal{F}(f_\theta(C, I))$, where E is the retrieved evidence annotated by humans as relevant.

If both hypotheses are supported, it would show the usefulness of the re-written queries and the retrieved evidence generated by our pipeline in im-

[5] https://huggingface.co/neulab/codebert-python

[6] https://huggingface.co/datasets/codeparrot/github-code-clean

proving next-cell generation for Jupyter notebooks. To evaluate H1, H2 we use the following baselines:

• **InCoder-1B** (Fried et al., 2022): InCoder is a unified generative model that can perform left-to-right program synthesis and editing via masking and infilling. The model has been pre-trained on over 670k GitHub repos containing Python and Jupyter notebooks (with JuICe data filtered out). We follow the XML prompting format suggested by the authors for giving the notebook context C and the markdown intent I, or query Q as shown in figure 5. The target cell $\hat{A}$ is generated by infilling a code cell after Q or I. Code evidence is inserted as code cells and documentation as markdown cells before Q (as shown in fig 5). Due to prompt length constraints, we allow up to 5 previous cells of C+E.

• **GPT-3.5 (text-davinci-003)** (Ouyang et al., 2022): GPT-3.5 is a 175B parameter LLM which has been fine-tuned through Reinforcement Learning from Human Feedback (RLHF) to increase alignment with the intent of human users. It has few shot-learning and code-completion abilities. To adapt it to the Jupyter Notebook completion we convert markdown cells to comments and keep code cells as is for the context (C), while Q or I is treated as the final comment. For the evidence-augmented setting, we insert documentation as code comments and code as is before Q as shown in figure 6. To avoid prompt length issues we allow up to 10 cells of context and up to 5 evidence blocks. While CODEX (code-davinci-003) (Chen et al., 2021b) would have been more suitable for this task, OpenAI has closed access to it.

6 Results

We analyze each stage of the pipeline separately using a mixture of automated metrics and human annotations (for query re-writing and evidence retrieval). We also analyze the performance of simple baselines on the val+test split of our dataset, using CodeBLEU (Ren et al., 2020) and API-call overlap.

6.1 Query Generation:

We perform query-rewriting over the val+test split (JVal, JTest, JTrainS) of our data and come up with the following automated metrics to test the quality of the rewrites:

KP EM: Corpus level exact keyphrase mentions in generated/re-written query Q.

KP>1 EM: Percent of Qs containing at least one exact keyphrase mention.

Task	Majority Vote	IAA		
		At-Y	At-A	Y-A
$I \rightarrow Q$	96%	65.75	46.81	26.04
$Q \leftrightarrow A$	78%	51.25	52.83	20.77

Table 1: Results of human annotation for the faithfulness of query Q to the original markdown I ($I \rightarrow Q$) and relevance of query Q to the target cell A ($Q \leftrightarrow A$), for queries Q generated from intent I by GPT-3.5

KP SBERT: SBERT score between keyphrases and Q to account for paraphrasing.

KP fuzzy: Relaxed version of KP EM based on fuzzy matching (`token_set_ratio` measure from fuzzywuzzy[7])

API EM, API>1 EM, API SBERT, API fuzzy: defined similarly to their KP counterparts.

The results for the automated evaluation metrics described above for JVal, JTest, and JTrainS are shown in table 7 for the rewritten queries generated by GPT-3.5. Based on the results there is better API coverage for instances in JTrainS (compared to JVal and JTest) but poorer exact keyphrase match. However, the KP SBERT score is higher indicating some paraphrased inclusion of the KPs, which can't be picked up by fuzzy matching.

We also perform annotations across the following two dimensions to manually evaluate the rewrites as shown in table 1.

6.2 Evidence Retrieval:

For the evidence retrieval phase, we evaluate the supervised bi-encoder code transformer approaches on their respective corpora (CodeSearchNet (Husain et al., 2019) or CoNaLa (Yin et al., 2018)) using standard IR metrics like MRR, NDCG, and recall@k as shown in tables 5 and 6. We also evaluate all the retrievers based on API coverage@10 (described below) and human-annotated relevance (binary), with the results being shown in table 2.

API coverage@k: Let C_k represent the top-k relevant codes retrieved by a retriever, A represents the target cell, and f be an AST-based function call extractor, that returns the set of unique function calls in a code fragment C as $f(C)$. We compute API coverage@k as: $API_{cov@k} = \frac{\sum_{a \in f(A)} I(a \in \bigcup_{C \in C_k} f(C))}{\|f(A)\|}$, where $I(x)$ is 1 if x is true and 0 otherwise. We allow for partial/suffix match while matching function calls.

[7] https://pypi.org/project/fuzzywuzzy/

Pipeline	Rertreiver	API Call Cov@10	Majority Vote	All Vote
code2code	**Code BM25**	**59.302**	**25.71**	**8.57**
	CoNaLa Ensemble	36.088	16.67	8.33
text2text	BM25	26.05	6.25	0
	DPR (TAS-B)	32.968	**39.29**	**7.14**
text2code	**Code SearchNet Ensemble**	**35.096**	**33.33**	**14.81**
	CoNaLa Ensemble	32.355	15.62	6.25
API lookup	**devdocs**		13.64	4.55

Table 2: API call coverage and annotator preference evaluation of evidence retrievers. We do not compute API call coverage for devdocs as it only contains documentation.

Majority vote: Percent of evidence retrieved by a retriever marked as relevant by two out of three annotators ($Q/A \leftrightarrow E$ annotation)
All vote: Same as above, but only considering evidence marked as relevant by all 3 annotators.

The results suggest that code2code pipeline is most effective in terms of covering API calls (since it focuses only on the target cells) and CodeBM25 is the best retriever according to API call coverage.

However, annotators prefer the text2text pipeline specifically the DPR model based on the majority vote. This trend shifts to the text2code pipeline when all votes (complete consensus between annotators) are considered.

6.3 KB Curation - Documentation Coverage

We check the coverage of APIs used in the target cells for their presence in the KB using exact match. Our documentation corpus has coverage of **92.83%** over API calls present in the sampled dataset.

6.4 Baselines Results for Hypotheses:

We test H1 and H2 using InCoder-1B and GPT-3.5. We evaluate the target cell generation in the following 3 settings:
CI: In this setting the code generator is expected to generate the target cell A, given the context C of previous markdown and code cells and a natural language intent I for generating the target cell.
CQ: In this setting the original markdown intent I is replaced with the re-written query Q. We expect this to improve the performance of code generators as Q conveys the intent of I in a more compact way and contains API call information. This would show that our queries are less ambiguous than the original intent.
CQE: In this setting, we add human-annotated gold evidence E to the context C and query Q. We hypothesize that our gold evidence are helpful in improving performance over the CQ setting.

We test hypothesis H1 by comparing performance differences between CI and CQ over the whole val+test data, the results for which are shown in table 3. For testing H2 we compare performance differences between CI, CQ, and CQE over an annotated subset of the val+test data (results shown in table 4). Results show that adding Q and E improves the notebook completion performance of the models.

7 Error Analysis

In this section, we present the results of qualitative analysis and error types of each stage of the pipeline as well as the CQ and CQE task settings (as highlighted by the chosen baselines).

7.1 Query Generation Errors:

Despite the high faithfulness score ($I \rightarrow Q$) obtained from our query generation approach, it can sometimes produce queries Q that contradict the original markdown intent I. For example for the markdown `There are some columns that we aren't too concerned with, so let's get rid of them. Specifically, **Major_code** means nothing to us Distinction between full time and part time isn't important right now, so let's drop **employed_full_time_year_round**`, the intent is to drop the columns `Major_code` and `employed_full_time_year_round`. However, the generated query Q is `Drop the columns **Major_code, Distinction,** and **employed_full_time_year_round** from the dataset.`, which misinterprets `Distinction` as a column name to be removed.

7.2 Evidence Retrieval Errors:

Based on the qualitative analysis, here are some of the errors:
Library Mismatch: The retrieval model returns

Setting	API Overlap	CodeBLEU	NGram Match	Weighted NGram Match	Syntax Match	DataFlow Match
CI	36.380	23.732	17.991	19.312	28.227	29.399
CQ	**40.088**	**24.107**	**18.347**	**19.441**	**29.089**	**29.551**

Table 3: H1: Comparing CI and CQ settings over val+test split of our dataset

Setting	Model	API Overlap	CodeBLEU	NGram Match	Weighted NGram Match	Syntax Match	DataFlow Match
CI	InCoder	36.960	26.314	19.83	21.68	32.71	31.03
CQ	InCoder	37.606	**29.124**	**23.4**	**24**	**34.11**	**34.98**
CQE	InCoder	**37.933**	25.774	20.25	21.2	31.03	30.61
CI	GPT-3.5	43.953	43.095	27.2	39.72	52.28	**53.17**
CQ	GPT-3.5	49.651	42.81	28.33	38.54	**52.75**	51.62
CQE	GPT-3.5	**55.168**	**44.123**	**29.92**	**42.37**	52.10	52.47

Table 4: H2: Comparing CI, CQ, and CQE settings over an annotated subset of val+test data

the code snippets that have same API calls but sourced from under different library. The figure 8 in appendix shows an example.

API function mismatch: There is overlap in imports and variables/ function names. However, the API coverage is low or zero. Refer to 9

Algorithmic mismatch: As shown in 9 correct APIs are called but their sequence or logic is far off. Mathematical and reasoning questions show such errors.

Definitions in notebook context: The target cells refer to blocks, functions and variables declared in Jupyter notebook context. Retrieval won't help in such scenarios. Refer to 11 in appendix.

7.3 Modeling Errors and Task Setting:

For H2 we observed that adding evidence decreases the CodeBLEU score for InCoder but increases it for GPT-3.5. A possible explanation is a difference in the prompting strategies followed. Since GPT-3.5 has a larger context window we can add the evidence while having 10 cells of context, but the context is replaced by the evidence for InCoder. This leads to cases like the one shown in figure 7 where evidence that is less relevant than the context replaces it leading to the generated code diverging from the target. Thus ignoring evidence when the context is more relevant is a possible issue that modeling approaches might need to consider.

8 Conclusion

This study proposed a dataset construction pipeline for long context open domain code QA (NBQA), and shows that reformulating the markdown intents to queries and evidence retrieval boosts code generation abilities and helps in personalization it to the long context Jupyter notebooks. We show that our query generation pipeline is scalable, generates good quality and faithful reformulations of their corresponding markdown intents. In terms of KB curation, the study suggested ensuring good coverage for top imports, both in the long and fat tail. Finally, the retrieval pipeline ensures diversity in the retrieved evidences. Overall, the proposed dataset is a good first step towards open domain long context code QA.

9 Future Directions

In this work, we have proposed a novel dataset for long context code completion in an open-retrieval setting. We have shown that our approach can outperform existing methods on two benchmark datasets. However, there are still some potential future directions for this work. One direction is to annotate the rest of the validation and test sets, which would require simplifying the annotation workflow and using context information inspired by conversational QA. Another direction is to add data from ExeDS and ARCADE to the validation and test sets, which would improve the data quality and enable execution-based evaluation. A third di-

rection is to try more baselines, such as fine-tuning on the noisy train set, training retrievers, and using FID style fine-tuning for the rewrite step.

References

Rajas Agashe, Srini Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *ArXiv*, abs/1910.02216.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Yang Wang. 2020. HybridQA: A dataset of multi-hop question answering over tabular and textual data. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1026–1036, Online. Association for Computational Linguistics.

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

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *ArXiv*, abs/2204.05999.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *ArXiv*, abs/2203.03850.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366.

Sebastian Hofstätter, Sheng-Chieh Lin, Jheng-Hong Yang, Jimmy Lin, and Allan Hanbury. 2021. Efficiently teaching an effective dense retriever with balanced topic aware sampling. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, page 113–122, New York, NY, USA. Association for Computing Machinery.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin B. Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *ArXiv*, abs/2211.09374.

Anette Hulth. 2003. Improved automatic keyword extraction given more linguistic knowledge. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, EMNLP '03, page 216–223, USA. Association for Computational Linguistics.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436.

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

Gautier Izacard and Edouard Grave. 2020. Leveraging passage retrieval with generative models for open domain question answering.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org.

Mayank Kulkarni, Debanjan Mahata, Ravneet Arora, and Rajarshi Bhowmik. 2022. Learning rich representation of keyphrases from text. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 891–906, Seattle, United States. Association for Computational Linguistics.

Dong Bok Lee, Seanie Lee, Woo Tae Jeong, Donghwan Kim, and Sung Ju Hwang. 2020. Generating diverse and consistent QA pairs from contexts with information-maximizing hierarchical conditional VAEs. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 208–224, Online. Association for Computational Linguistics.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks.

Haau-Sing Li, Mohsen Mesgar, André F. T. Martins, and Iryna Gurevych. 2022. Asking clarification questions for code generation in general-purpose programming language. *ArXiv*, abs/2212.09885.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation.

Chenxiao Liu and Xiaojun Wan. 2021. CodeQA: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2618–2632, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 1287–1293. AAAI Press.

Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. Clcdsa: Cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1026–1037.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. 2022. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016b. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aur'elien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971.

Svitlana Vakulenko, Shayne Longpre, Zhucheng Tu, and Raviteja Anantha. 2020. Question rewriting for conversational question answering.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1693–1703. International World Wide Web Conferences Steering Committee.

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

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, A. Eashaan Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. 2022. Natural language to code generation in interactive data science notebooks. *ArXiv*, abs/2212.09248.

Hamed Zamani, Fernando Diaz, Mostafa Dehghani, Donald Metzler, and Michael Bendersky. 2022. Retrieval-enhanced machine learning. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation.

Mengxiao Zhang, Yongning Wu, Raif Rustamov, Hongyu Zhu, Haoran Shi, Yuqi Wu, Lei Tang, Zuohua Zhang, and Chu Wang. Advancing query rewriting in e-commerce via shopping intent learning. *The 2023 SIGIR Workshop On eCommerce*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs.

A Limitations

Despite the benefits of our proposed dataset, there are still some limitations and challenges that are needed to be addressed. One of the limitations is the dependency on the Python version and the library version that are used in the code demonstrations. Different versions may have different syntax, features, or behaviors that can lead to failures or errors in the execution. Another limitation is the coverage of the libraries that are supported by our systems. These libraries form a long and fat tail in the distribution and needs additional filtering or indexing. A third limitation is the potential training data leakage happening when Large Language Models are pretrained on GitHub. This may give an unfair advantage to the LLMs over other models, and might require additional obfuscation of the validation and test sets. A fourth limitation is the granularity of indexing that we use to from the knowledge base. It is an open question if Class-Level or Method-Level indexing is optimal for the code snippets.

B Ethical Considerations

Our work on code question answering has some social implications that need to be examined and addressed. A major concern is the privacy and security of code repos. Private information, API keys, etc. shouldn't be indexed and licensing and intellectual properties should be respected. Another implication is the linguistic biases that may arise from our dataset collection, sampling, and modeling approaches. Since our dataset is primarily focused on English, it may not reflect the diversity and richness of other languages used by programmers globally and may limit the applicability and accessibility of our systems for non-English speakers. Additionally, biases in annotator selection and in annotation may affect the quality and validity of the labels and the evaluation metrics.

C Computational Costs

The QA systems in this project have significant computational costs in terms of model and GPU size. The carbon footprint of the systems includes the training of code models for retrievals and the inference of LLAMA, GPT-3.5, InCoder, and retrieval models. The GPU usage involves A4500 and A6000 cards, which have an estimated emission of 19.44 kg of CO2 for training the retriever. Moreover, the annotation pipeline is slow and requires annotation of multiple evidences per query through MTURK evaluation. A simplificaiton of this pipeline is necessary to ensure cost-efficiency of the annotation to scale the annotation.

D Deployment Challenges

Deploying a QA system in the real world poses several challenges that need to be addressed. One of the main challenges is running inference at scale, which requires efficient and robust hardware and software solutions. Another challenge is performing retrieval from large and dynamic corpora, which may affect the speed and accuracy of the system. Moreover, the quality of the data that the system relies on is not always in control, and may contain noise, errors, or inconsistencies. Finally, large language models that are often used for QA systems may suffer from hallucination, biases, or data leakage, which can compromise the reliability and trustworthiness of the system.

E Reflection

This work helped us understand the code question answering task and the dataset design challenges. The main challenge is data curation, which involves many steps and choices. For instance, we had to select an LLM and a prompt for generating queries from contexts. We also had to create a knowledge base that is diverse and heterogeneous, covering various libraries. Additionally, we had to retrieve evidences from code repositories that match the

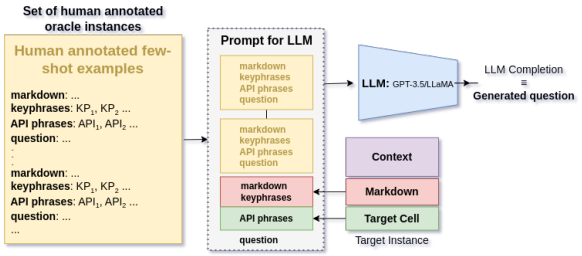

Figure 3: The prompting pipeline used for query generation. We use a set of three fixed few-shot examples enriched with the keyphrase and API phrase information. We also leverage keyphrases and API phrases for the target intent I which is to be re-written to obtain Q

queries. This required choosing and ensembling models for retrieval to obtain the best results. Moreover, we had to devise a simple, consistent, and accurate annotation standard. Another challenge was data cleaning, which removed duplicates, and outliers in the data. We had to ensure the data quality and validity. A third challenge was data preprocessing, which involved formatting or transforming the data for training and evaluation. We had to deal with different data types, such as code, markdown, and queries.

F Contribution

Aditya: Query Generation (LLaMA), Text2Text+Text2Code Retrieval, KB Curation, Annotation
Atharva: Query Generation (GPT-3.5), Key Phrase Extraction, Baselines, Text2Code+Code2Code Retrieval, Annotation
Yash: DevDocs, Text2Code, Code2Code Retrieval, KB Curation, Annotation

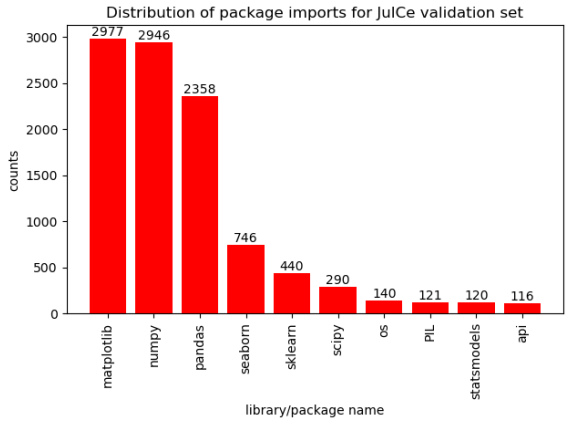

Figure 4: Distribution of imports for various packages in the JuICe validation set consisting of 1.8k notebooks. Some packages are imported through aliases which we collapse here leading to counts for some packages being greater than the number of notebooks.

```
<text>
step 2 : Generating a sample of size n of the multivariate normal distribution ...
</text>
<cell>
n=5000
mu=[0,0]
sigma = [[5,5],[5,10]]
alpha_0=0.01
degreesFreedom=len(mu)
mormalizerMatrix, eigenValues, eigenVectors=mvNormalizerMatrix(sigma)
</cell>
<text>
matplotlib _as_gen matplotlib pyplot figure: Create a new figure, or activate an
existing figure.
</text>
<cell>
def plot_conf(fignum, s, datablock, pars, new):
    """
    plots directions and confidence ellipses
    """
    ...
    #
    # put on the mean direction
    #
    x, y = [], []
    XY = pmag.dimap(float(pars[0]), float(pars[1]))
    x.append(XY[0])
    y.append(XY[1])
    plt.figure(num=fignum)
    ...
    #
    # plot the ellipse
    #
    plot_ell(fignum, pars, 'r-,', 0, 1)
</cell>
<text>
Step 3 Visualizing the sample and the constant density ellipse for differents leves of
confidence $(1-\\alpha)$%:
</text>
<code>
</code>
```

Figure 5: Prompting format used for InCoder-1B. We give up to 5 previous NB cells (purple highlight), while the code evidence (blue highlight) and doc evidence (orange highlight) is inserted before the markdown I/query Q (in bold). XML tags are used to distinguish markdown and code (following (Fried et al., 2022)). The answer is infilled at the green highlight.

Model	Val				Test			
	MRR	R@1	R@5	R@10	MRR	R@1	R@5	R@10
CodeBERT	**54.79**	**41.65**	73.84	85.5	60.9	49.32	79.18	87.95
CodeBERT-Python	54.31	41.13	73.43	85.21	**62.55**	**50.96**	**79.73**	87.4
GraphCodeBERT	54.05	39.97	**75.35**	**86.6**	61.58	50.14	78.63	**88.22**
UniXcoder	53.76	41.18	72.91	83.99	56.5	46.3	75.07	84.11

Table 5: Results of dense bi-encoder transformer-based retrievers after contrastive training (same procedure as (Husain et al., 2019)) on the CoNaLa dataset

Model	Val				Test			
	MRR	R@1	R@5	R@10	MRR	R@1	R@5	R@10
CodeBERT-Python	71.78	61.92	84.36	89.27	**73.5**	**64.05**	**85.6**	90.1
GraphCodeBERT	73.55	63.86	**85.89**	**90.6**	75.3	65.9	87.37	**91.41**
UniXcoder	**75.3**	**66.12**	**87.01**	**91.06**	**76.05**	**67.12**	**87.41**	**91.65**

Table 6: Results of dense bi-encoder transformer-based retrievers after contrastive training (same procedure as (Husain et al., 2019)) on the CodeSearchNet dataset

split	KP EM	KP>1 EM	KP SBERT	KP fuzzy	API EM	API>1 EM	API SBERT	API fuzzy
JVal	**72.25**	93.53	38.89	**80.71**	53.83	80.07	39.05	60.72
JTest	71.97	**94.07**	39.21	80	50.41	**80.93**	38.75	57.77
JTrainS	64.26	91.84	**42.84**	75.17	**57.24**	77.18	**41.99**	**65.86**

Table 7: Results of automatic evaluation of re-written queries for GPT-3.5

Martkdown	Question
It looks like more people from Group B turned in an application. Why might that be? We need to know if this difference is statistically significant. Choose a hypothesis tests, import it from scipy and perform it. Be sure to note the p-value. Is this result significant?	Find if the difference between the number of people from Group B who turned in applications is statistically significant, using scipy hypothesis testing and print the p-value.
Word2Vec model - Build a Word2Vec model from sentences in the corpus. - Set the maximum distance between the current and predicted word within a sentence to 10. - Ignore all words with total frequency lower than 6.	gensim models word2vec question: Create a word2vec model using gensim, ignoring words with total frequency lower than 6 and maximum distance of 10 between current and predicted words.
Where would we be without Kepler? The Kepler Space Telescope is the 800-pound gorilla in the room, but there's more to exoplanet discovery than just Kepler. It wasn't even launched until 2009, and there are almost 50 other facilities that have found confirmed exoplanets. What happens if we ignore Kepler?	Plot a bar graph using the data from the other 50 facilities that have found confirmed exoplanets, ignoring Kepler, and label the x-axis, y-axis, and title accordingly.

Table 8: Examples of original markdowns I and the question Q that captures the intent of I while including API call information about the target cell A, to reduce ambiguity

```
# step 2 : Generating a sample of size n of the multivariate normal distribution ...

n=5000
mu=[0,0]
sigma = [[5,5],[5,10]]
alpha_0=0.01
degreesFreedom=len(mu)
mormalizerMatrix, eigenValues, eigenVectors=mvNormalizerMatrix(sigma)

# matplotlib _as_gen matplotlib pyplot figure: Create a new figure, or activate an
existing figure.

def plot_conf(fignum, s, datablock, pars, new):
      """
      plots directions and confidence ellipses
      """
      ...
      #
      # put on the mean direction
      #
      x, y = [], []
      XY = pmag.dimap(float(pars[0]), float(pars[1]))
      x.append(XY[0])
      y.append(XY[1])
      plt.figure(num=fignum)
      ...
      #
      # plot the ellipse
      #
      plot_ell(fignum, pars, 'r-,', 0, 1)

# Step 3 Visualizing the sample and the constant density ellipse for differents leves of
confidence $(1-\\alpha)$%:
```

Figure 6: Prompting format used for GPT-3.5 (text-davinci-003). We give up to 10 previous NB cells (purple highlight), while the code evidence (blue highlight) and doc evidence (orange highlight) is inserted before the markdown I/query Q (in bold). Markdown cells are supplied as comments while code is left as is. The answer is the continuation predicted by the model.

Code-BM25: A BM-25-based approach, that uses AST analysis to extract variable names, function calls, and imported module names from code to create a bag of words document representation

Figure 7: A potential modeling issue for our dataset (CQE setting): If the evidence is less relevant than the notebook context, approaches sensitive to recent context or with limited context windows (like InCoder) might suffer from the addition of evidence close to the target markdown I. The replaced context in the CQ setting is highlighted in purple and the doc evidence and code evidence that replace it are highlighted in orange and blue. The evidence added causes divergence from the gold answer cell.

```
def fit_and_predict(X_train, y_train, X_test, random_state):
    '''
    Fits Decision Trees.

    Parameters
    __________
    X: A pandas.DataFrame. Training attributes.
    y: A pandas.DataFrame. Truth labels.

    Returns
    _______
    A numpy array.
    '''
    ##################
    # YOUR CODE HERE
    dtc = tree.DecisionTreeClassifier(random_state=random_state)
    dtc.fit(X_train, y_train)

    prediction = dtc.predict(X_test)
    ##################

    return prediction
```

```
def predict(self, x):
    """
    Predict values for a single data point or an RDD of points using
    the model trained.

    .. note:: In Python, predict cannot currently be used within an RDD
        transformation or action.
        Call predict directly on the RDD instead.
    """
    if isinstance(x, RDD):
        return self.call("predict", x.map(_convert_to_vector))

    else:
        return self.call("predict", _convert_to_vector(x))
```

Figure 8: Error case of code retrieval. We see that the function name "predict" is present in different libraries.

```python
def calculate_accuracy(predicted, actual):
    correct_guesses_count = np.sum(predicted == actual)
    return correct_guesses_count / len(actual)
```

```python
import numpy as np
from sklearn import preprocessing
X = np.array(['cat', 'dog', 'cow', 'cat', 'cow', 'dog'])
binar = preprocessing.LabelBinarizer()
X_bin = binar.fit_transform(X)
print X_bin

[[1 0 0]
 [0 0 1]
 [0 1 0]
 [1 0 0]
 [0 1 0]
 [0 0 1]]
```

Figure 9: Error case of code retrieval. Although both query and result have functions of the same NumPy library, API overlap is zero..

```python
### Include your answer here (you can use math.gamma if needed)
def get_kurtosis():
    # Return a list with 4 numbers / expressions

    Kurt_1 = 2
    Kurt_2 = -0.5
    Kurt_3 = pow(math.gamma(1/4), 2) / (4 * pow(math.gamma(3/4), 2)) - 3
    Kurt_4 = 6

    return [Kurt_1, Kurt_2, Kurt_3, Kurt_4]
```

```python
def beta(a, b):
  """use gamma function or inbuilt math.gamma()
  to compute vals of beta func"""
  beta = math.exp(math.lgamma(a)
    + math.lgamma(b) - math.lgamma(a + b))
  return beta
```

Figure 10: Error case of code retrieval. The query expects a different mathematical logic.

```python
def summarize_by_class(X, y):
    '''
    Separates a training set into instances grouped by class.
    It then calculates the summaries for each attribute.

    Parameters
    __________
    X: A 2d numpy array. Represents training attributes.
    y: A 1d numpy array. Represents class labels.
    Returns
    _______
    A dictionary of 2d numpy arrays
    '''
    ##################
    # YOUR CODE HERE
    summaries = {}
    X_separated = separate_by_class(X, y)
    for key in X_separated:
        summaries[key] = summarize(X_separated[key])
    ##################

    return summaries
```

```python
def _generate_classes(self, class_types, non_defined, **kwargs):
    """ creates the class for each class in the data set

        args:
            class_types: list of class_types in the dataset
            non_defined: list of subjects that have no defined class
    """
    # kwargs['dataset'] = self
    for class_type in class_types:
        self[class_type[self.smap]] = self._get_rdfclass(class_type,
                                                         **kwargs)\
                                                        (class_type,
                                                         self,
                                                         **kwargs)

        self.add_rmap_item(self[class_type[self.smap]],
                           class_type[self.pmap],
                           class_type[self.omap])

    for class_type in non_defined:
        self[class_type] = RdfClassBase(class_type, self, **kwargs)
        self.add_rmap_item(self[class_type], __a__, None)
    self.__set_classes__
    try:
        self.base_class = self[self.base_uri]
    except KeyError:
        self.base_class = None
```

Figure 11: Error case of code retrieval. The function "summzarize" is defined in the Jupyter notebook context".