Learning to Edit Interactive Machine Learning Notebooks

Bihui Jin*
bihui.jin@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Jiayue Wang*
jiayue.wang@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Pengyu Nie
University of Waterloo
Waterloo, Ontario, Canada
pynie@uwaterloo.ca

Abstract

Machine learning (ML) developers frequently use interactive computational notebooks, such as Jupyter notebooks, to host code for data processing and model training. Notebooks provide a convenient tool for writing ML pipelines and interactively observing outputs. However, maintaining notebooks, e.g., to add new features or fix bugs, can be challenging due to the length and complexity of the ML pipeline code. Moreover, there is no existing benchmark related to developer edits on notebooks.

In this paper, we present early results of the first study on learning to edit ML pipeline code in notebooks using large language models (LLMs). We collect the first dataset of 48,398 notebook edits derived from 20,095 revisions of 792 ML-related GitHub repositories. Our dataset captures granular details of file-level and cell-level modifications, offering a foundation for understanding real-world maintenance patterns in ML pipelines. We observe that the edits on notebooks are highly localized. Although LLMs have been shown to be effective on general-purpose code generation and editing, our results reveal that the same LLMs, even after finetuning, have low accuracy on notebook editing, demonstrating the complexity of real-world ML pipeline maintenance tasks. Our findings emphasize the critical role of contextual information in improving model performance and point toward promising avenues for advancing LLMs' capabilities in engineering ML code.

CCS Concepts

• **Software and its engineering → Software evolution**; • **Computing methodologies → Machine learning**.

Keywords

Interactive computational notebook, machine learning pipeline, software evolution, code editing, large language model

ACM Reference Format:
Bihui Jin, Jiayue Wang, and Pengyu Nie. 2025. Learning to Edit Interactive Machine Learning Notebooks. In *Companion Proceedings of the 33rd ACM Symposium on the Foundations of Software Engineering (FSE'25), June 23–27, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728523

*Both authors contributed equally to this research.

FSE Companion'25, Trondheim, Norway

ACM ISBN 979-8-4007-1276-0/2025/06
https://doi.org/10.1145/3696630.3728523

1 Introduction

The widespread adoption of interactive computational notebooks as a development platform in machine learning (ML), particularly Jupyter notebooks [9], has made them essential for developing, documenting, and debugging ML pipelines [4, 5, 20–22]. Despite utility, *maintaining* notebooks presents significant challenges, such as frequent and fragmented edits, especially when the notebooks grow in size and complexity.

Software development and maintenance have been significantly transformed by the advancement of large language models (LLMs) [7, 12, 14, 18], offering an alternative for developers to efficiently manage repetitive and time-consuming code edits [10]. As notebooks typically interleave domain-specific code on a specific topic (e.g., machine learning) and narrative text, it is cumbersome to apply rule-based program analyses techniques to infer developer intent and automate code editing. Thus, data-driven approaches like LLMs can be ideal solutions for learning to edit code in notebooks.

Existing datasets on notebooks focus primarily on code generation and comprehension tasks [1, 5, 16, 20–22]. They do not track developers' code edits or revisions, limiting their relevance for understanding and automating the maintenance of notebooks.

To bridge the gap, we propose a novel task of *notebook editing*: automatically editing code in an interactive computational notebook given the developer change intent. In this study, we establish a testbed for the notebook editing and evaluate the performance of the state-of-the-art LLMs on the task. The unique code structure of notebooks poses a challenge for notebook editing compared to general-purpose code editing. Specifically, notebooks are composed of interleaving code cells, text cells, and execution outputs; an edit can be localized in one cell or span across multiple cells.

Since no existing dataset exists for this task, we present the first notebook editing dataset with 48,398 Jupyter notebook edits, mined from 20,095 revisions across 792 ML-related GitHub repositories. Our dataset captures detailed insights into the evolution of notebooks, offering a granular perspective on the maintenance practices of real-world ML pipelines. We capture both file-level and cell-level edits, along with commit messages to represent developer change intents. Notably, while on average each repository has 8,380 lines of code, the majority of developer edits target specific portions of notebooks, with an average of 166 lines modified per revision.

To motivate further study on this new direction, we evaluate the performance of code LLMs on our notebook editing dataset. Our experiments are performed based on a state-of-the-art code LLM, DeepSeek-Coder [7], and explore both in-context learning and supervised finetuning. Surprisingly, even with finetuning, the LLMs fail to perform correct edits on notebooks at either file-level or cell-level, despite the base LLM's effectiveness in general-purpose code tasks [7]. This highlights the challenges in automating real-world

notebook edits and suggests that combining LLMs with execution and agentic frameworks may be necessary for this task.

The main contributions of this paper include:

- We propose notebook editing, a novel task for automatically editing ML pipelines in notebooks given developer change intent.
- We collect a dataset of 48,398 Jupyter notebook edits, capturing both file-level and cell-level edits with commit metadata.
- We benchmark state-of-the-art code LLMs with few-shot learning and supervised finetuning and find that notebook editing remains challenging for LLMs.
- Based on analyses of our dataset and early experiment results, we discuss next steps towards improving LLMs' performance on the notebook editing task.

Our dataset and experiment scripts are open-sourced at https://github.com/uw-swag/ipynb-edit.

2 Study Design

2.1 Dataset Construction

Figure 1 shows an overview of the our dataset construction steps, which consists of three phases: fetching GitHub repositories, processing data, and filtering data.

First, we identify a list of GitHub repositories that contain notebooks with ML pipelines. In this work, we focus on Jupyter notebook [9]. We utilize the GitHub Search API [6] to query the GitHub repositories that are tagged with "jupyter-notebook" and "machine-learning" topics. We keep the top 1,000 repositories, sorted based on the popularity (determined by the number of stars).

Then, we clone each repository to extract and process data, specifically through the following steps:

(1) retrieve the commit history using `gitlog`, and record the hash and commit message for each entry that modifies at least one Jupyter notebook (with `.ipynb` file extension);
(2) fetch the content of the notebook file before and after each edit using `git show`;
(3) use the `SequenceMatcher` functionality in Python's difflib library to obtain the code difference (diff) between the old and new versions of the notebook file, both at cell level (within each file) and at line level (within each changed cell).

Note that we focus on the code cells in the Jupyter notebooks, and discard the text cells (which are for documentation purposes) and execution outputs (which may frequently change unintentionally due to rerunning). The collected dataset includes the repository name, commit hash, file name, commit message, old and new versions of code, cell-level diff, and line-level diff. After processing, we only retain Python code in the notebooks and discard the text (markdown) cells and execution outputs.

Finally, we refine the dataset to ensure the quality of our dataset. Specifically, we exclude the entry whose commit messages contain two or fewer words and remove duplicate entries.

2.2 LLM Experimental Setup

2.2.1 Models. We use **DeepSeek-Coder** [7] as the base LLM in our study, which is a pretrained decoder-only transformer model tailored for code-related tasks. Specifically, we use the instruction-finetuned version with **1.3B** and **6.7B** parameters.

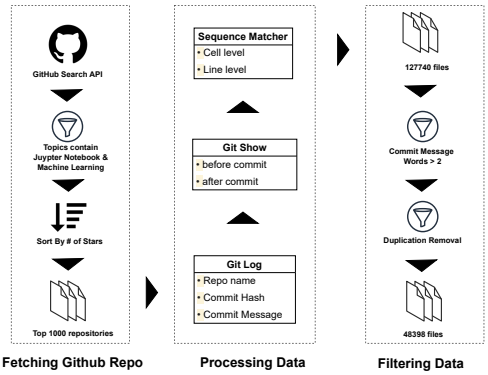

Figure 1: An overview of data construction workflow.

2.2.2 Scenarios and Inputs/Outputs. We design two scenarios of completing the notebook editing task:

Cell-level prediction. The developer selects which part (a subset of cells) of the notebook to edit. The LLM is given the inputs of the commit message and the old version of the selected cells, and is expected to output the new version of those cells. In the experiment, we presume the selected cells to be the ground-truth subset of cells changed by developers.

File-level prediction. The LLM is given the inputs of the commit message and the old version of notebook (with all cells concatenated, delimited by <cell_#></cell_#> tags, where # is the cell number), and is expected to output the new version of the entire notebook.

In both scenarios, the inputs to the LLM are embedded in a prompt that (1) lets the LLM impersonate the role of an expert software developer with an instruction, (2) describes the inputs and outputs of the given task, and (3) supplies few-shot examples.

2.2.3 Learning Strategies. We explore two distinct strategies to train the LLM on the notebook editing task:

In-context learning. LLMs can learn to perform an unseen task with a number of few-shot examples in the prompt [3]. In-context learning does not incur any training cost but requires the task to be similar to the pretraining task. We provide five few-shot examples randomly sampled from the training set.

Supervised finetuning. With additional finetuning on the training set, LLMs can better learn to perform the task and adapt to the input/output format required by the task. We adopt a parameter-efficient finetuning technique, LoRA [8]. Due to the high cost incurred in training, we only perform finetuning on the 1.3B model.

2.2.4 Environment and Hyper-Parameters. We conduct our experiments on a server with AMD EPYC 7343 CPU (16 Cores @ 1.5GHz), Nvidia RTX A6000 GPU (48G RAM), running Ubuntu 22.04 LTS. To account for randomness, we repeat each of our finetuning experiment three times and report the average results. Our model inference is deterministic since we adopt greedy decoding.

3 Results

3.1 Dataset Statistics and Analysis

3.1.1 Quantitative Analysis. Table 1 shows the size of our notebook editing dataset after splitting into training/validation/testing sets. Table 2 presents statistics including commit message length, file/cell

Table 1: Size of our notebook editing dataset after splitting.

Set	#Proj	#File	LOC	#Commits	#ΔCells	#ΔLines
full	792	48,398	6,636,971	20,095	508,634	3,336,142
train	536	27,515	3,510,884	12,874	308,731	1,899,031
val	76	12,469	1,988,162	3,660	116,313	970,452
test	152	8,414	1,137,925	3,561	83,590	466,659

Table 2: Statistics of our notebook editing dataset.

Name	Avg	Min	Med	75%	90%	Max
commit message #words	5.66	3	5	7	9	72
file #token before edit	1,511.60	1	593	2,092	4,191	50,451
file #token after edit	2,079.28	1	1,260	2,756	4,845	79,893
changed cells #token before edit	344.14	1	1	204	944	30,800
changed cells #token after edit	841.14	1	191	947	2,437	77,597
changed #cells per edit	29.73	1	11	26	56	16,969
changed #lines per edit	173.56	1	55	151	371	153,689
changed #token per edit	2,577.88	2	761	2,209	5,424	2,188,763

sizes before/after edits, and the number of cells/lines/tokens per diff. We summarize key characteristics of our dataset below.

Notebook edits generally contain more code addition than removal. The files show an average of 1,511.60 tokens pre-edit and 2,079.28 tokens post-edit, which suggests a focus on incremental development and the inclusion of additional content during updates.

The size of edits varies and can be very large. We observe an extremely long-tail distribution of code edit size in our dataset, ranging from single-line or single-cell edits to 153K lines or 17K cells of edits. Larger code edits can be significantly harder to predict, as they may exceed LLMs' context length.

Developer edits in notebooks focus on localized areas. By comparing the average number of tokens in files (1,511.60–2,079.28) and the average number of tokens in the changed cells (344.14–841.14), we observe that only a part of the cells are changed in each version. We find that the median number of cells changed is 11, and in many cases, only one cell is touched in a code edit.

Developers tend to write very brief commit messages. Commit messages have only 5.66 words on average, with 75% containing up to 7 words (even after filtering non-descriptive commit messages of two or fewer words). This brevity poses a challenge in understanding developer change intents purely from commit messages.

3.1.2 Qualitative Analysis. To qualitatively understand the content of notebook edits, we analyze the frequent tokens in the commit messages and code edits (excluding symbols and stop words).
Commit messages. Our key finding is that **developers prioritize adding new functionalities over fixing or removing, with a trend on performing edits at the notebook or file level**. The word "add" appears 12,083 times, significantly more than "update" (7,484), "fix" (4,309), or "remove" (2,571), showing developers frequently increment their ML pipelines with new functionalities and components. Although "fix" and "remove" are secondary to "add", their combined frequency (6,880) indicates that nearly 57% as much effort is dedicated to maintenance and refinement compared to adding new features. Such balance shows that while new features dominate, significant resources are allocated for ensuring quality and removing redundant elements. In addition, the frequent occurrence of structural terms like "notebook" (7,910) and "file" (6,488) hints that developers frequently orchestrate edits at the file level.
Code edits. Comparing the commit messages between the code addition and deletion, we find that **the code edits predominantly reflect iterative development of ML workflows, including frequent adjustments to dependencies, data handling, modularity, and cloud deployment configurations**. In particular, "import" is the most frequently deleted word (49,271) and added word (156,220), indicating that most changes tend to rely on modifying model's dependencies. Additionally, high frequency of domain-specific terms, such as "model" (35,907 added / 11,687 removed), "dataset" (14,425/5,120), "train" (14,189/4,832), and "test" (14,109/4,475), largely pertain to iterative adjustments to ML models, datasets, and associated processes.

3.2 LLM Evaluation Results

3.2.1 Metrics. To evaluate the accuracy of LLMs in performing code edits, we use several code similarity metrics that are frequently used in prior work on LLM for code generation [13, 23] to compare the predicted new code against reference (developer-written) new code, including: **BLEU** [15], which computes the percentage of overlapping $1 \sim 4$-grams; **CodeBLEU** [17], which extends BLEU for code-specific tasks by incorporating syntax and data flow; **Edit similarity** (**EditSim**) [19], defined as 1 minus Levenshtein edit distance; and **RougeL** [11], which uses the F1-score between subtoken matches based on the longest common subsequence.

These similarity metrics don't capture functional correctness when the generated code differs from the reference code. For file-level prediction, we measure the percentage of **Executable** files without runtime errors. We use a docker environment provided by Kaggle with popular ML libraries installed.[1] Of the 8,414 files in our test set, 1,134 (13.74%) are executable in our environment, and we only measure Executable on this subset following prior work [13]. Models may generate trivial but executable files (e.g., with only import statements), therefore we additionally measure **LOC** as a proxy for whether the generated code is non-trivial. Future work should develop better functional correctness metrics verifying both executability and the correctness of execution results.

3.2.2 Results and Findings. Table 3 and Table 4 compare the performance of DeepSeek-Coder models on cell-level and file-level notebook editing, respectively. The first two rows are 1.3B and 6.7B models with in-context learning; the next two rows (with "-postproc" suffix) attempt to fix the formatting errors in the outputs (e.g.., removing redundant code block symbols or repeated instructions) to improve the performance of in-context learning; the last row shows the results of supervised finetuned 1.3B model.

All models, even after supervised finetuning, face difficulties on the notebook editing task, reflecting the complexity of the task. In-context learning with 1.3B or 6.7B model achieves low accuracy, which is expected, as the size of the edits can be large and hard to learn within LLMs' limited context window. In fact, we observe that in-context learning sometimes fails to follow our desired input/output formats and mistakenly outputs redundant code block tokens or repeating a part of prompts, and our attempt to address these problems with postprocessing results in moderate performance improvements. Supervised finetuning 1.3B achieves the best performance, with a CodeBLEU score of 30.55 on cell-level prediction and 45.26 on file-level prediction. However, this does

[1] https://gcr.io/kaggle-gpu-images/python. Setting up correct execution environments for each repository and revision is challenging, which we leave as future work.

Table 3: Cell-level prediction results.

Model	BLEU	CodeBLEU	EditSim	RougeL
1.3B	8.52	17.26	26.52	21.49
6.7B	13.30	20.49	31.01	28.05
1.3B-postproc	8.59	18.12	26.64	21.66
6.7B-postproc	13.44	22.06	31.23	28.30
1.3B-finetune	**25.86**	**30.55**	**46.53**	**40.80**

Table 4: File-level prediction results. The functional correctness metrics (Executable and LOC) are measured on the subset of files where developer-written files are executable.

Model	BLEU	CodeBLEU	EditSim	RougeL	Executable	LOC
1.3B	10.75	15.39	31.78	27.23	73.66	97.49
6.7B	11.76	15.34	34.68	30.45	15.17	**772.51**
1.3B-postproc	10.78	17.42	31.76	27.36	**87.83**	81.97
6.7B-postproc	11.81	18.64	34.70	30.57	25.49	676.77
1.3B-finetune	**27.46**	**45.26**	**47.38**	**53.13**	34.07	720.20

not meet the demands of real-world code editing. Considering that the same model, DeepSeek-Coder, has been shown to be effective on many other code generation and editing tasks, our finding hints that notebook editing may be one of the most challenging tasks.

LLMs perform better on file-level editing than cell-level editing, where the former one involves more contextual code than the latter one. We initially reckon that the additional code context included in file-level prediction may distract the LLM, and knowing the edit locations (as in cell-level prediction) would give advantages to the LLM. Nevertheless, the results support the opposite direction: the finetuned 1.3B model achieves 50% higher CodeBLEU on file-level prediction than cell-level prediction. This means that contextual information in the rest of the notebook file may be important, e.g., for the LLM to learn coding styles and examples. Note that models predict Python code rather than JSON raw format of the notebook in our experiment.

LLMs are weak at performing functionally correct notebook edits. From the last two columns of Table 4, we observe that although 87.83% of the files generated by the 1.3B-postproc model are executable, the average LOC is only 81.97, which is much lower than the average LOC of reference files (577.87), meaning that the generated files are likely not functionally correct (i.e., they are executable but not generating the expected execution results). Using larger models or performing finetuning improves LOC and potentially leads to more meaningful code (as suggested by the similarity metrics), but the generated files usually contain runtime errors. For example, the 1.3B-finetune model only has 34.07% executable rate.

4 Discussions and Future Work

As the first work in this direction, our experiments are still limited, namely, only one type of the LLM (DeepSeek-Coder) and two sizes of LLMs (1.3B, 6.7B) are used, and functional correctness metrics (e.g., checking if the edited notebook is executable) are lacking. The short length of commit messages in our dataset indicates that they may not be a good representation of developer change intents, as supported by prior studies [2]. Future work may supplement the extraction of developer change intents with other sources, e.g., issue trackers and pull requests data.

At this stage, we avoid excessive data filtering so that we can comprehensively understand the needs of notebook editing in the real-world data. We find that the code edits in notebooks are diverse in terms of size, intents, and topics. Leveraging these insights, future work can explore subsets that focus on specific types of code edits.

We have shown that notebook editing can be an extremely challenging task for pure LLM solutions. Based on our experience, we plan to explore retrieval-augmented-generation and agentic techniques to improve the performance of LLMs on predicting notebook edits because extracting relevant contextual information is critical in performing targeted edits in usually lengthy notebooks.

5 Related Work

LLMs have been utilized in various software engineering tasks [1, 5, 16, 20–22]. In particular, it has demonstrated promise in code generation within Jupyter notebooks [1, 22]. These datasets underscore the capacity of LLMs to leverage natural language and code content for context-based code generation and maintenance tasks. Nonetheless, it is not yet clear the ability of their benchmarks to longer contexts in Jupyter notebooks, especially when the code depends on prior cells or external resources, which may not generalize well to unseen or non-typical notebook styles. Studies [5, 16] provide insights into Jupyter notebooks code metrics to understand and improve computational notebook quality. Despite these advancements, key challenges remain, such as understanding developers' code editing behaviors and improving LLMs' performance on tasks involving real-world maintenance scenarios in Jupyter notebooks. To this, Wang et al. [21] leverage run-time information for ML bug detection in Jupyter notebooks by incorporating dynamic data into static analysis methods, which underscores a promising direction for integrating static and dynamic analyses to address challenges in ML code quality. In our study, we propose a dataset of 48,398 Jupyter notebooks edits from 20,095 revisions across 792 ML repositories, capturing cell-level and file-level changes, along with commit metadata, explore the capability of LLMs for suggesting code edits in interactive ML notebooks, and demonstrate the complexity of our dataset in representing real-world ML maintenance tasks.

6 Conclusions

We present the first study on learning to edit ML pipelines in Jupyter notebooks. By mining 20,095 revisions of 792 open-source ML-related GitHub repositories, we collect a dataset of 48,398 notebook edits. Our dataset reveals the diversity of notebook editing intents and the challenging size of edits for LLMs to automate. To motivate further research on notebook editing, we evaluate LLMs' ability to predict both file-level and cell-level edits. Even with supervised fine-tuning, LLMs exhibit low accuracy on our dataset, highlighting the complexity of real-world ML maintenance tasks.

Acknowledgments

We thank Saikat Dutta, Owolabi Legunsen, Kaiyuan Wang, and the anonymous reviewers for their comments and feedback. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference number RGPIN2024-04909.

References

[1] Rajas Agashe, Srini Iyer, and Luke Zettlemoyer. 2019. JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation. In *Conference on Empirical Methods in Natural Language Processing*.

[2] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *International Conference on Software Engineering*. 134–144.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[4] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *International Conference on Software Engineering*. 995–1007.

[5] Mojtaba Mostafavi Ghahfarokhi, Arash Asgari, Mohammad Abolnejadian, and Abbas Heydarnoori. 2024. DistilKaggle: A Distilled Dataset of Kaggle Jupyter Notebooks. In *International Working Conference on Mining Software Repositories*. 647–651.

[6] GitHub. 2022. REST API endpoints for repositories. https://docs.github.com/en/rest/repos?apiVersion=2022-11-28

[7] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence.

[8] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *International Conference on Electronic Publishing*.

[10] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.

[11] Chin-Yew Lin and Franz Josef Och. 2004. Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics. In *Annual Meeting of the Association for Computational Linguistics*.

[12] Microsoft. [n. d.]. Search Microsoft Copilot: Your everyday AI companion. https://copilot.microsoft.com/

[13] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. *International Conference on Software Engineering* (2023), 2111–2123.

[14] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[15] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Annual Meeting of the Association for Computational Linguistics*.

[16] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2021. KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle. In *International Working Conference on Mining Software Repositories*. 550–554.

[17] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *ArXiv* abs/2009.10297 (2020).

[18] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[19] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020).

[20] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael J. Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2021. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *Transactions on Computer-Human Interaction* 29 (2021), 1–33.

[21] Yiran Wang, Jose Antonio Hernandez Lopez, Ulf Nilsson, and Daniel Varro. 2024. Using Run-Time Information to Enhance Static Analysis of Machine Learning Code in Notebooks. In *Companion Proceedings of the International Symposium on the Foundations of Software Engineering*. 497–501.

[22] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. 2023. Natural Language to Code Generation in Interactive Data Science Notebooks. In *Annual Meeting of the Association for Computational Linguistics*. 126–173.

[23] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-evolution using Large Language Models. *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023).